Learning Timed χ 1.0

Preliminary version, please send feedback and remarks to: a.t.hofkamp@tue.nl

J. Vervoort and J.E. Rooda

Augustus, 2007

Technische Universiteit Eindhoven, Department of Mechanical Engineering
Systems Engineering Group, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
http://se.wtb.tue.nl/

Contents

1 Introduction **1**

2 My first χ-specification **5**

2.1 Exercise(s) 6

3 A simple process **7**

3.1 Multiple statements 8

3.2 Multi-assignment 8

3.3 Standard output and standard input 8

3.4 Initialization 9

3.5 Exercise(s) 10

4 Data types and operators **11**

4.1 Boolean variables and operators 11

4.2 Natural variables and operators 12

4.3 Integer variables and operators 14

4.4 Real variables and operators 14

4.5 String variables 15

4.6 Using data types 15

4.7 Defining custom types 16

4.8 Exercise(s) 17

5 Statements **19**

5.1 Skip statement 19

5.2 Sequential composition 20

5.3 Alternative composition 20

5.4 Guard operator 21

5.5 Loop statement . . . 25
5.6 While statement . . . 26
5.7 Operator Priorities . . . 29
5.8 Exercises . . . 31

6 Communicating processes **35**
6.1 Two processes . . . 35
6.2 Multiple instantiations of a process . . . 38
6.3 Parametric processes . . . 39
6.4 A process of processes . . . 40
6.5 Parallel machines . . . 42
6.6 Exercises . . . 43

7 Timed Processes **47**
7.1 Simulating time . . . 47
7.2 Exercises . . . 49

8 Stochastic behaviour **53**
8.1 Exercises . . . 56

9 Functions **59**
9.1 Simple functions . . . 59
9.2 Functions versus processes . . . 61
9.3 Function calls within functions . . . 62
9.4 Higher order functions . . . 64
9.5 Exercises . . . 66

10 Lists **71**
10.1 Basic properties . . . 71
10.2 Concatenation and subtraction . . . 71
10.3 Basic list functions . . . 72
10.4 Take and drop . . . 76
10.5 Sorting lists . . . 78
10.6 Mapping and filtering lists . . . 80
10.7 Folding lists . . . 81
10.8 Exercises . . . 82

11 Record tuples **87**

11.1 A record tuple 87

11.2 Multi-assignment 88

11.3 Record tuples in functions 88

11.4 Lists of record tuples 90

11.5 Exercises 92

12 Vectors **95**

12.1 A vector 95

12.2 Exercises 99

Bibliography **103**

A ASCII Conversion **105**

A.1 Two conversion examples 105

A.2 ASCII conversion tables 106

Chapter 1

Introduction

Manufacturing systems become more and more complex and the requirements on these systems increase. In addition, the manufacturing system has to be built in a shorter period of time. As a consequence, the design of such systems also becomes more demanding and requirements on the design process increase. In the past, manufacturing systems were often designed by trial-and-error and based on empirical knowledge. Nowadays, modelling and simulation plays a prominent role in the design process.

An essential aspect of the system complexity is concurrency: concurrency of multiple machines, buffers, transportation devices, as well as concurrency of multiple machine components. Examples of concurrent manufacturing systems contain both factories and machines: a semiconductor factory, a beer brewery, a luggage handling system, an automated warehouse, an electrical component placer, a clinical chemical analyzer, or a waferstepper.

The desire grew for a modelling and simulation tool that:

- can easily describe parallelism in manufacturing systems,
- can describe manufacturing systems by a discrete-event model, and
- can be used by manufacturing system engineers with little computer science background.

Origin of χ

Research on the specification language χ for modelling, simulation, and control of concurrent manufacturing systems started some twenty years ago by Rooda [Roo82]. The specification language χ is a result of years of cooperative research by several people. A detailed overview of the various contributions can be found in the acknowledgements at the end of this chapter. The specification language χ incorporates elements from process algebra [Fok00][BM02] and communicating sequential processes [Hoa95]. Moreover, it contains elements from functional programming [BW88]. Finally, its formal semantics are defined by process algebra [BM06]. The specification language χ is developed as a modelling and simulation tool for design of concurrent systems, such as embedded systems and manufacturing systems. However, other

areas of application can be thought of, such as computer networks, or traffic(control). In the sequel, we discuss the key characteristics of χ and a brief consideration of the consequences of these characteristics.

Key characteristics of χ

Parallel processes

χ-specifications consist of parallel processes. This allows decomposition of complex systems into several parallel (less complex) subsystems. This hierarchical decomposition provides the modeller with an intuitive way to break down system complexity, and thus to represent large complex systems, even entire factories.

Short symbolic notation

Contrary to most general purpose programming languages, χ uses mainly symbols and a few keywords. This accentuates the structure in models and improves readability.

Small set of high-level elements

χ has a small set of high-level elements suited for describing communication. Within the application domain, specifications can be short and no elaborate programming background is required. Functionality such as statistics, matrix calculations, graphics, animation, user-interfacing, or using data-bases, can be provided with by external programs Matlab [Mat02], Python [Lut99], Labview [Nat01], or SQL [SQL02] that can be coupled to a χ-specification using Python.

Stochastics

In manufacturing systems stochastic behaviour plays an important role. χ has a powerful random generator that is extended with the possibility to simulate most commonly encountered distributions.

Formal semantics

The semantics of χ are formally described by a process algebra. In principle, this allows for models to be verified mathematically. With verification all possibilities are investigated, contrary to simulation, where one possible path is evaluated.

Executable models

Not all specification languages allow for simulating a model. χ-specifications can be compiled to obtain an executable program, which in turn behaves like the model.

Real-time control

We can use χ-specifications to design and implement real-time control. χ-specifications can be compiled to a program, that runs on a real-time platform, such as real-time Linux.

On this tutorial

The language concepts are mainly introduced by simple examples. The examples are kept domain independent as much as possible. At several instances, we describe a case from practice to illustrate the practical value of certain language concepts. Each chapter contains a number of exercises to operationalize the introduced language concepts. This tutorial is based on χ1.0.

For more on using χ in a particular domain, we refer to domain-specific lecture notes:

- Analysis of Manufacturing Systems (4C530),
- Analisis of Hybrid Systems (4C650),
- Supervisory Machine Control (4K420), and
- Engineering Optimization: Concepts and Applications (4J530).

Overview

In Chapters 2 and 3 we present the first simple χ-specifications. Subsequently, the different basic data types are discussed in Chapter 4. The different basic constructs are discussed in Chapter 5. Chapters 6 and 7 deal with concurrent processes, one of the most important characteristics of χ, and simulating time. Chapter 8 discusses ways to describe stochastic behaviour in χ-specifications. Chapter 9 deals with functions, higher-order functions, and recursive functions. Chapters 10 through 12 deal with the compound data types list, records, and vectors.

Acknowledgements

Credits for the design of the language go to all members of the χ-club. In particular, we would like to mention the following contributions. J.M. van de Mortel-Fronczak stood at the base of the language [MR95]. N.W.A. Arends was the first to describe the formalism [Are96]. W. Alberts and G.A. Naumoski wrote the first compiler (χ0.3) [AN98]. A.T. Hofkamp wrote the real-time compile system [Hof01b] and the compilers. V. Bos and J.J.T. Kleijn helped in providing a formal framework for discrete event χ 0.8 [BK02]. Later hybrid extensions were added by K.L. Man and R.R.H. Schiffelers [MS06]. The compiler was made by A.T. Hofkamp [HR06] and hybrid simulator by R.R.H. Schiffelers. The cooperation of the Systems Engineering Group with the Parallel Programming Group (formerly lead by M. Rem) and the Formal Methods Group (lead by J.C.M. Baeten) has contributed to this language

We thank W.A.P. van den Bremer, MSc student of the Systems Engineering Group, for adapting this manuscript for χ 1.0.

Chapter 2

My first χ-specification

Below is a simple χ-specification.

proc $P() =$ |[!!"`Chi, it works deliciously!`"]|

model $M() =$ |[$P()$]|

This specification consists of one process in which the string "Chi, it works deliciously!" is written to the screen. The last line defines the χ model and the identifier(M), that represents the name of the model. Each specification requires a definition of a model in order to run.

In order to execute (simulate) this specification, you should type this specification in a ASCII file editor such as PFE or WinEdt under Windows, or VIM or VI under Linux/Unix. The symbolic notation has to be translated into ASCII. Note that chi is case-sensitive. Below, the ASCII equivalent of the symbolic specification is shown.

```
proc P() = |[ !!"Chi, it works deliciously!" ]|

model M() = |[ P() ]|
```

In this simple example the ASCII equivalent is straightforward. However, as we will see furtheron some symbols have no straightforward ASCII equivalent. Appendix A contains a complete overview of ASCII equivalents for different χ-elements. After converting the symbolic notation into ASCII, you save the model with the extension .chi (here `exmp21.chi`). Then you can compile it by typing the following.

```
[user@host chi]$ chic exmp21.chi
```

The $-sign represents the Linux prompt. In the Linux distribution that is used for running the examples in this tutorial, the prompt is preceded by `[user@host chi]`. The word `chic` is an acronym for chi compiler. The compiler generates an executable model with the same filename as it's original, but with no extension. This executable model can be executed by typing the command `startmodel` followed by the name of the model in the Linux command

prompt. The command `startmodel` actually starts the simulation of the model. Simulating this model yields:

```
[user@host chi]$ startmodel exmp21
Chi, it works deliciously![user@host chi]$_
```

2.1 Exercise(s)

1. Type, compile and run the example presented in this chapter.

Chapter 3

A simple process

A χ-specification is built around processes. For now we consider specifications containing a single process only. Remember that when you want to compile and simulate a specification you need to add a model definition: `model M() = |[ P() ]|`, see Chapter 2.

A process has the following general form:

```
proc P() =
|[ declarations
:: statements
]|
```

Herein proc stands for process, not for procedure. The process name is P, here P has no parameters. The brackets () are mandatory. Furtheron, we will see what parameters a process can have. The opening bracket (|[) denotes the start of the process and the closing bracket (]|) denotes the end of the process. In the beginning of the process we declare all variables that are used. A separator (::) separates the declarations from the actual process, which consists of statements. The complete syntax is described using railroad diagrams in the χ reference manual [HR06].

Consider the following example:

```
proc P() =
|[ var i: nat
:: i:= 2
]|
```

The keyword *var* is used to indicate that the process variables are declared. This process uses one variable, i, of type natural $\mathbb{N}$ (0,1,2,3,..). Other data types are described in Chapter 4. The process contains one statement: an assignment statement. The variable i is assigned the value 2; one says: i becomes 2. As we will see in Chapter 4, this is different from i equals 2 ($i = 2$).

3.1 Multiple statements

It is possible to perform a sequence of statements. To this end, we separate the individual statements by semicolons.

```
proc P() =
|[ var i,j: nat
:: i:= 2; j:= 3
]|
```

Here, two variables are used, i and j, both of type natural. The process contains two statements. The statements are executed sequentially. First, variable i is assigned the value 2, then variable j is assigned the value 3. In this example the statements are placed after each other on the same line. It is also allowed to place statements under each other on separate lines, as is done in the specification below.

```
proc P() =
|[ var i,j: nat
:: i:= 2
 ; j:= 3
]|
```

3.2 Multi-assignment

In the previous section two sequential statements were used to assign values to variables i and j. It is also possible to use one assignment statement to assign values to both i and j simultaneous.

```
proc P() =
|[ var i,j: nat
:: i,j:= 2,3
]|
```

3.3 Standard output and standard input

It is possible to write data to the screen (standard output) by means of two exclamation marks (!!) . Consider the following example:

```
proc P() =
|[ var i,j: nat
:: i:= 2
 ; !!"i has the value ", i, "\n"
 ; j:= 3
 ; !!"j has the value ", j, "\n"
]|
```

Compiling and executing this specification (exmp34.chi) yields:

```
[user@host chi]$ startmodel exmp34
i has the value 2
j has the value 3
[user@host chi]$_
```

The string that is written to the standard output in this example is built using several substrings. The substrings are separated by commas (,). Literal text is to be placed between double quotes ("). Variables are not placed between quotes. The string "\n" stands for new line. Its effect is similar to pressing the return-key. The string "\t" stands for tabulator stop. Its effect is similar to pressing the TAB-key as is shown in the next example.

It is possible to read from the standard input (keyboard) by using double question marks (??).

```
proc P() =
|[ var i: nat
:: !!"Enter a natural number: "; ??i
 ; i:= i + 1
 ; !!"i=\t", i, "\n"
]|
```

Executing this specification (exmp35.chi) yields:

```
[user@host chi]$ startmodel exmp35
Enter a natural number
3
i =     4
[user@host chi]$_
```

First the string "Enter a natural number " is written to the screen. Then a natural number is requested from the standard input and assigned to the variable i, here 3 was entered. Subsequently 1 is added to i. The final value of i is written to the screen. The '=' and the '4' are separated by a tabulator stop.

3.4 Initialization

So far we declared a variable and then assigned a value to it with an assignment statement. It is possible to do this more efficiently. We declare a variable and immediately give it a value. This is called *initialization*.

```
proc P() =
|[ var i: nat = 0
:: i:= i + 1
]|
```

The first line in the variable declaration should be read as: variable i is of type nat and is initially equal to 0.

In a similar way it is possible to initialize multiple variables of the same type.

```
proc P() =
|[ var i,j: nat = (1,2), k: nat
:: k:= i + j; !!k, "\n"
]|
```

Here variable i is initialized with the value 1 and j is initialized with the value 2.

3.5 Exercise(s)

1. Execute the following specification.

```
proc P()=
|[ var fname,lname: string
:: !!"What's your first name:\n"; ??fname
 ; !!"What's your last name:\n"; ??lname
 ; !!"Hello ", fname, " ", lname, "! \n"
]|
```

2. Write a process that prompts the user to enter a natural number i and a natural number j and writes the sum of these two to the screen.

3. Below are two processes (P and Q) that are designed to swap the values of variables x and y. Explain which process successfully swaps the values. Verify its functionality by execution and adding statements that write output to the screen.

```
proc P() =
|[ var x,y: nat = (1,10)
:: x:= y; y:= x
]|

proc Q() =
|[ var x,y: nat = (1,10), h: nat
:: h:= x; x:= y; y:= h
]|
```

Chapter 4

Data types and operators

As we have seen in Chapter 3, each process consist of a declaration part. Herein, each variable that is used in the process is declared, and it's type is defined. The type of the variable determines the range of values which it may take. Each data type has a specific set of operations that can be performed on values of that type. The following basic data types are distinguished:

data type		example values
bool	$\mathbb{B}$	false, true
nat	$\mathbb{N}$	0, 1, 5, 423
int	$\mathbb{Z}$	-43, -90, +0, +10, +82
real	$\mathbb{R}$	3.452, -1.11, 18.0, 3.5e-10
string		"Systems Engineering", "Chi, it works deliciously!"

Besides these basic data types compound data types, such as lists, record tuples, vectors, and sets exist. These compound data types are discussed in Chapters 10 through 12.

In the sequel, the possible operators for each data type are discussed.

4.1 Boolean variables and operators

Boolean variables are of the type bool. The term bool stems from the English logician and mathematician George Boole (1815-1864) as he developed the boolean algebra. Within this algebra a variable can have only two possible values: false or true. We distinguish the following operators on boolean values.

$\neg$	not
$\wedge$	and
$\vee$	or

The negate operator $\neg$, has the highest priority, followed by the 'and' operator $\wedge$, followed by the 'or' operator $\vee$. For example, $\neg p \vee q \wedge r$ will be read as $(\neg p) \vee (q \wedge r)$, in which p,

q and r are boolean values. We can also say $\neg p \vee q \wedge r \equiv (\neg p) \vee (q \wedge r)$. The equivalence symbol ($\equiv$) denotes that the expression on the left side of this symbol is equivalent to the expression on the right side of it.

We first explain the negate operator. This is a unary operator, i.e. it operates on a single variable. If p is false, then $\neg p$ will be true. If p is true, then $\neg p$ will be false. We can write this in a truth table. We obtain the following:

p	$\neg p$
false	true
true	false

The 'and' operator is also called the conjunction operator. Again we can write this result in a truth-table:

p	q	$p \wedge q$
false	false	false
false	true	false
true	false	false
true	true	true

In the same way, we can define the binary-operator 'or'. The 'or' operator is also called the disjunction operator. We have the following truth table:

p	q	$p \vee q$
false	false	false
false	true	true
true	false	true
true	true	true

Below, is an example of a χ-specification that uses boolean variables and operators.

```
proc P() =
|[ var p: bool = true, q: bool = true, r: bool = true
:: !!"p = ", p, "\tq = ", q, "\tr = ", r, "\n"
 ; !!"not p or q and r = ", not p or q and r, "\n"
 ; !!"(not p) or (q and r) = ", (not p) or (q and r), "\n"
]|
```

This example shows that for p = true, q = true, and r = true, the boolean expression $\neg p \vee q \wedge r$ is equivalent to $(\neg p) \vee (q \wedge r)$.

4.2 Natural variables and operators

Variables of the type nat can have values from the set of natural numbers $\mathbb{N}$. This is the collection of all whole non-negative numbers including zero, e.g. 0, 3, 4, and 71. In χ the following binary operators, i.e. operators requiring two arguments, are defined on natural numbers.

x^y	raising to the power
$x \times y$	multiplication
x div y	quotient of natural division
x mod y	remainder of natural division
$x + y$	addition
$x - y$	substraction

The addition and substraction operators have the lowest priority. Raising to the power has the highest priority.

When we write 'a div b' we determine the number of times that b fits in a, e.g. 5 div $3 = 1$. The remainder of the division is calculated by 'a mod b', e.g. 5 mod $3 = 2$. Consider the following χ-specification that performs a number of operations on natural values.

```
proc P() =
|[ var i: nat = 3, j: nat = 5
:: !!"i + j = ", i + j, "\n"
 ; !!"j div i = ", j div i, "\n"
 ; !!"i div j = ", i div j, "\n"
 ; !!"j mod i = ", j mod i, "\n"
 ; !!"(j div i) * i + j mod i = "
    , (j div i) * i + j mod i, "\n"
]|
```

Compiling and running this specification (exmp43.chi) yields:

```
[user@host chi]$ startmodel exmp43
i + j = 8
j div i = 1
i div j = 0
j mod i = 2
(j div i) * i + j mod i = 5
[user@host chi]$_
```

There also exist relational operators: $<$, $>$, $\leq$, $\geq$, $=$, $\neq$. These operators have a lower priority than the addition and substraction operator. The result of this operation is a boolean, e.g. $2 = 2 \equiv$ true, $2 > 3 \equiv$ false. Consider the following example:

```
proc P() =
|[ var i: nat = 2, j: nat = 4, b,c: bool
:: b:= i < j; c:= i = j
]|
```

Note the essential difference between the assignment symbol (:=) and the equality symbol (=).

4.3 Integer variables and operators

Variables of the type int can have values from the set of integers, $\mathbb{Z}$, which are all whole numbers. Integers are whole numbers preceded by a plus (+) or a minus (−) sign. To denote the number zero as an integer, it should be preceded by a plus or a minus sign. The following numbers are integer numbers: −31, −12, +0, and +67.

For integers the same operators as for the naturals are available. In addition, there exists the unary negation operator minus (−). Consider the following example:

```
proc P() =
|[ var i: int = +2, j,k: int, b,c: bool
:: j:= -i; k:= i + j
 ; b:= i < j; c:= i = j
 ; !!i, "\t", j, "\t", k, "\n", b, "\t", c, "\n"
]|
```

Compiling and running this specification (exmp44.chi) yields:

```
[user@host chi]$ startmodel exmp44
+2      -2      +0
false   false
[user@host chi]$_
```

4.4 Real variables and operators

Real numbers (reals) are denoted by the keyword real. Real numbers are always presented with at least one decimal. Real numbers can also be written in exponential notation. Examples are: 2.0, 45.5, -27.9, 3e-10. For real variables the unary negation operator minus (−) is defined, similarly as for integers. The relational operators discussed for the naturals, are also available for reals. Moreover the following binary operators are defined:

x^y	raising to the power
$x \times y$	multiplication
x/y	division
$x + y$	addition
$x - y$	subtraction

The binary operators + and − have the lowest priority. The unary operator − has the highest priority, followed by the ^ (raising to the power) operator. Consider the following example:

```
proc P() =
|[ var x: real = 3.0, y,z,q: real
:: y:= 5.0/3.0
 ; z:= 3e-1 / 5.0
 ; q:= 1.0 - 10 * 0.1
 ; !!x, "\t", y, "\n", z, "\t", q, "\n"
]|
```

Compiling and running the specification (exmp45.chi) yields:

```
[user@host chi]$ startmodel exmp45
3.00000000000000000     1.66666666666666674
0.06000000000000000     -0.00000000000000006
[user@host chi]$_
```

As can be seen from the output real values are imprecise. In this case they are only precise up till 16 decimals. This is a familiar problem when using computers for calculations.

4.5 String variables

Variables of type string represent a sequence of characters. A string is always enclosed by double quotes. Examples are “Systems Engineering” and “Chi, it works deliciously!”. Strings can be composed from different strings, separated by commas, as discussed in Chapter 3. For strings there is a binary operator, called the concatenation operator (++) to add one string after another.

Moreover, the relational operators $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$ can be used for strings as well. In the case of string variables they compare variables alphabetically, e.g. “a” $<$ “aa”, “aa” $<$ “ab”, “ab” $<$ ”b”, ”aa” $=$ ”aa” and ”aa” $\neq$ ”ab”. Consider the following example:

```
proc P() =
|[ var s,u: string = ("a","b"), t,v: string
:: t:= s ++ "a"; v:= s ++ u
 ; !!"s = ", s, "\t t = ", t, "\t u = "
   , u, "\t v = ", v, "\n"
 ; !!"s < t is equivalent to ", s < t, "\n"
 ; !!"t < u is equivalent to ", t < u, "\n"
 ; !!"u < v is equivalent to ", u < v, "\n"
]|
```

Compiling and running the specification (exmp46.chi) yields:

```
[user@host chi]$ startmodel exmp46
s = a    t = aa  u = b   v = ab
s < t is equivalent to true
t < u is equivalent to true
u < v is equivalent to false
```

4.6 Using data types

χ uses a strict data typing, data types have to be used consistently. The sequel contains some examples, in which a value is assigned to a variable of a non-corresponding type. Compiling this model will result in a “type conflict” error message.

```
proc NotAllowed() =
|[ var x: real, i: int, j: nat // it is not allowed to
:: x:= 3   // assign a natural value to a real variable
 ; i:= 0.0 // assign a real value to an integer variable
 ; j:= +3  // assign an integer value to a natural variable
]|
```

The text preceded by double forward slashes (//) is interpreted as a comment. In the following process the variable typing conventions are correctly obeyed.

```
proc Allowed() =
|[ var x: real, i: int, j: nat // it is allowed to
:: x:= 3.0 // assign a real value to a real variable
 ; i:= +0  // assign an integer value to an integer variable
 ; j:= 3   // assign a natural value to a natural variable
]|
```

It is allowed to add, subtract, multiply and divide variables of different types. The type of the result can differ from the original types. Consider the following examples:

operation	type change
$3 + +4 = +7$	nat, int $\rightarrow$ int
$3.0 + 4 = 7.0$	real, nat $\rightarrow$ real
$+3 \times 4.0 = 12.0$	int, real $\rightarrow$ real
$x = 3,\ y = +x$	nat $\rightarrow$ int

In general, one could say that the variable range of the result is a superset of the variable ranges of the original variables. A complete overview of the resulting type when using operators (+,-,×,/) on different types can be found in the χ reference manual.

4.7 Defining custom types

It is possible to define your own types. Below, the type lot is defined as a nat and the type product is defined as a bool.

```
type lot     = real
   , product = bool

proc P() =
|[ var x: lot, p: product
:: x:= 4.0
 ; p:= false
]|
```

As we will see in Chapter 11, defining your own data types is especially useful when using compound data types, it improves readability.

4.8 Exercise(s)

1. Try compiling and running the specification below. Study the error message(s) and correct the error(s).

```
proc P() =
|[ var x: real = 4.0, y: real = -2, a,b: bool
:: a:= x > y
 ; b:= (x - y) > 0
 ; !!x,"\t",y,"\n"
 ; !!"x > y is equivalent to ",a,"\n"
 ; !!"x - y > 0 is equivalent to ",b,"\n"
]|
```

2. Try compiling and running the specification below. Study the error message(s) and correct the error(s).

```
proc P() =
|[ var x: nat = 4.0, y: nat = 2, a,b: bool
:: a = x > y
 ; !!x,"\t",y,"\n"
 ; !!"x > y is equivalent to ",a,"\n"
]|
```

3. Write a process that prompts the user for a natural number, then squares the number and displays the result to the screen.

4. Write a process that prompts the user for a password. If the password equals a predefined string, the boolean variable *access* becomes true, otherwise *access* remains false. Print the results to the screen.

5. Write a process that evaluates the following two boolean expressions: $p \vee q$ and $\neg(\neg p \wedge \neg q)$ for any given p and q. Verify that both expressions are equivalent for all possible values of p and q. You can use part of the specification below:

$\chi-$ 4.1:

```
proc P() =
|[ var p: bool = ... , q: bool = ...
:: !!"p or q = ", ...
 ; !!"
]|
```

Chapter 5

Statements

In the previous chapters we have seen that a process can contain multiple statements. Statements in χ can be divided in two classes: the atomic statements, that represent the smallest statement units; and the compound statements, that are constructed from one or more (atomic) statements using operators.

We distinguish the following atomic statements:

- skip statement
- assignment statement

In the previous chapters we have shown the assignment statement.

We have the following compound statements:

- sequential composition
- guard operator
- alternative composition
- loop statement
- while statement

In the next chapters we will introduce some more atomic and operators.

5.1 Skip statement

Skip is an empty statement; it does nothing. Consider the following specification:

```
proc P() =
|[ skip ]|
```

This process does nothing. The skip statement might seem useless at this point, but as we will see furtheron, it can be useful in certain situations.

5.2 Sequential composition

In a sequential composition, statements are performed one after another. The previous chapters contain numerous examples. Below is an example from the exercises from Chapter 3.

```
proc Q() =
|[ var x: nat = 1, y: nat = 10, h: nat
:: h:= x; x:= y; y:= h
]|
```

In a sequential composition, statements are separated by a semicolon (;). In this example first the assignment `h:= x` is executed, followed by `x:= y` and finally `y:= h`.

5.3 Alternative composition

Alternative composition presents the possibility to choose between two or more alternatives. Consider the following example:

```
proc P() =
|[ var i: nat
:: i:= 1 | i:= 2
]|
```

This example will either assign the value 1 to variable i or it will assign the value 2 to variable i.
There are no parentheses used in this specification. However using them improves readability. Leaving them out would in this case yield the same result. Using parentheses and writing each alternative on a new line results in:

```
proc P() =
|[ var i: nat
:: ( i:= 1
   | i:= 2
   )
]|
```

Because using parentheses improves understanding and readability, we will use this way of writing from now on.

The general form of an alternative composition, containing n alternative statements S is as follows:

(S_0
| S_1
| ...
| S_n
)

Which alternative is executed, is chosen *non-deterministically*. Non-deterministic choosing means that the choice outcome is unknown. In other words the result cannot be determined, thus every result is possible. For instance, repeatedly non-deterministic choosing between alternatives S_0 and S_1 can yield $S_0, S_0, S_0, S_0, S_0, \ldots$ but can just as well yield $S_1, S_0, S_1, S_0, S_1, \ldots$ or $S_1, S_1, S_1, S_0, S_1, S_0, \ldots$. Notice that this is different from choosing randomly.

5.4 Guard operator

A guarded statement consist of a guard b, which is a boolean expression and a statement S, that is executed if this guard has the value true. It looks like this:

$$b \rightarrow S$$

Consider the following example:

```
proc P() =
|[ var b: bool = true
:: b -> !!"YES"
]|
```

Compiling and running the specification (guard.chi) yields:

```
[user@host chi]$ startmodel guard
YES
```

The guard b has the value true and the word YES is written to the screen.

Now lets see what happens if the guard is false:

```
proc P() =
|[ var b: bool = false
:: b -> !!"NO"
]|
```

Compiling and running this specification(guard1.chi) yields:

```
[user@host chi]$ startmodel guard1
At time 0, program halted (infinite delay)
```

The guard b is false, so the statement after the `->` is never executed. This is exactly indicated by the output. The program halts, because it can only wait forever.

Selection

We can now combine the guard operator and the alternative composition to form a selection to choose between two or more alternatives. Consider the following example:

```
proc P() =
|[ var b: bool = true
:: (     b -> !!"YES"
   | not b -> !!"NO"
   )
]|
```

In this composition, the first alternative is chosen if the boolean b has the value true (which is the case here). If b would have been false, then *not* b would result in true. This means that the second alternative would have been chosen. Another example is:

```
proc P() =
|[ var i: nat = 3, j: nat = 4
:: ( i >= j -> !!i
   | i < j  -> !!j
   )
]|
```

In this example, two variables (i and j) are used. If i is greater than or equal to j ($i \geq j$), the value of i is printed on the screen. If i is smaller than j ($i < j$), the value of j is printed on the screen. In other words, the maximum of i and j is printed to the screen.

Now take a look at the next example:

```
proc P() =
|[ var i: nat = 3, j: nat = 3
:: ( i >= j -> !!i
   | i <= j -> !!j
   )
]|
```

In this case both guards evaluate to true, $3 \geq 3$ and $3 \leq 3$. So either alternative can be chosen, again, in a *non-deterministic* fashion. So it will either choose $!!i$ or $!!j$.

The general form of a selection is as follows:

$$
\begin{array}{l}
(\ b_0 \ \ \rightarrow S_0 \\
|\ b_1 \ \ \rightarrow S_1 \\
|\ \ldots \rightarrow \ldots \\
|\ b_n \ \ \rightarrow S_n \\
)
\end{array}
$$

Herein b_i, for $0 \leq i \leq n$, denotes a boolean expression (guard) and S_i a statement. Notice how a selection is composed by combining a guard operator and an alternative composition. The statement itself is called a selection statement. An operational interpretation of the selection statement is as follows:

Upon execution of a selection all guards are evaluated. As long as none of the guards evaluates to true, the selection waits until one of the guards becomes true. If only one of the guards evaluates to true and its corresponding statement can be executed, then this statement is executed. If more than one of the guards evaluates to true and has a corresponding statement that can be executed, then one of these statements is chosen *non-deterministically* and executed.

Example 5.1 Determining the sign of an integer
The example process below prints the 'sign' of z to the screen.

```
proc P() =
|[ var z,s: int
:: !!"Enter an integer number:\n"; ??z
 ; ( z < 0  -> s:= -1
   | z = +0 -> s:= +0
   | z > 0  -> s:= +1
   )
 ; !!"Sign(z) = ", s, "\n"
]|
```

The user is prompted for an integer number. The entered value is assigned to the variable z. Then the string "Sign(z) = " is printed to the screen. Depending on whether z is less than, greater than, or equal to 0, "-1", "+0", or "+1" is printed to the screen. Finally a new line is added to the screen output. Compiling and running this example (exmp51.chi) yields (here -3 was entered):

```
[user@host chi]$ startmodel exmp51
Enter an integer number:
-3
Sign(z) = -1
[user@host chi]$_
```

⊠

After an alternative in a selection is chosen it is possible to execute more than one statement. It is even possible to have a selection statement (nested) within a selection. Consider the following example.

Example 5.2 Login procedure
This example will show a nested selection in a process that is used to log in using a username and password.

```
proc Login() =
|[ var un, pw: string, b: bool
:: !!"Enter username:\n"; ??un
 ; ( un = "guest" -> !!"Guest Access Granted"
   | un = "007"
     -> !!"Enter password:\n"; ??pw
      ; b:= pw = "opensesame"
```

```
        ; ( b     -> !!"Full Access Granted"
          | not b -> !!"Access denied: password invalid"
          )
    )
]|
```

The user must enter his username. If the username is "guest", the user is granted "Guest Access". If the user enters "007" for username, the user is asked to enter his password. The user is denied access unless he entered "opensesame" as password. ⊠

What happens when the user enters a username different from "guest" or "007", for instance "student"? Then execution of the process ends and the following message is printed to the screen "...program halted (infinite delay)". To avoid this kind of behaviour it is highly recommended to **cover all possible alternatives**. The example becomes:

```
proc Login2() =
|[ var un, pw: string, b: bool
:: !!"Enter username:\n"; ??un
 ; ( un = "guest" -> !!"Guest Access Granted"
   | un = "007"
     -> !!"Enter password:\n"; ??pw
      ; b:= pw = "opensesame"
      ; ( b     -> !!"Full Access Granted"
        | not b -> !!"Access denied: password invalid"
        )
   | un /= "007" and un /= "guest"
     -> !!"Acces denied: username invalid"
   )
]|
```

Note that, contrary to many programming languages, χ has no element comparable to 'else' as in 'if - then - else'. This forces the modeller to explicitly define the remaining alternatives. It might be the case that for some of these alternatives nothing should happen. Then the skip statement can be useful, as is demonstrated by the following specification:

```
proc P() =
|[ var i: int
:: !!"Enter an integer i: "; ??i
 ; ( i <  0 -> i:= -i
   | i >= 0 -> skip
   )
 ; !!"The absolute value of i = ", i, "\n"
]|
```

If i is less than 0, i becomes $-i$. If i is greater than or equal to 0, nothing happens (skip).

The use of boolean variables may result in some non-elegant specifications. The following specification is an example of non-elegant modeling.

```
proc P() =
|[ var b: bool
:: !!"Enter true/false:\n"; ??b
 ; ( b = true  -> !!"b is true"
   | b = false -> !!"b is false"
   )
]|
```

Why use '$b = \text{true}$' when b is already a boolean? A more elegant version, that behaves the same, would be:

```
proc P() =
|[ var b: bool
:: !!"Enter true/false:\n"; ??b
 ; (     b -> !!"b is true"
   | not b -> !!"b is false"
   )
]|
```

Though this may seem very obvious, this inelegancy is often encountered in practice!

5.5 Loop statement

The loop statement presents the possibility to execute a statement forever. The loop statement is indicated by a $*$ preceding the statement (S) that has to be repeated.

$*S$

Consider the following example:

```
proc P() =
|[ var i: nat = 0
:: *( i:= i + 1; !!"i = \t", i, "\n" )
]|
```

Variable i initially has the value 0. As the repetition is entered 1 is added to the current value of i. Then the new value of i is written as output to the screen. After that, both statements between the brackets are completed. The repetition then executes the two statements again, that is, 1 is added to i and i is written to the screen. This is repeated infinitely.

Notice that the use of parentheses is necessary here. Suppose we omit the parentheses.

```
proc P() =
|[ var i: nat = 0
:: * i:= i + 1; !!"i = \t", i, "\n"
]|
```

Now only the statement `i:= i + 1` is repeated forever. This is because of the priority of operators. Binding priority of operators is further explained in Section 5.7

5.6 While statement

As shown in the previous section, the repetition statement is repeated forever. It might also be the case that you want to repeat a statement a finite number of times. This can be done with the while statement, which repeats statement S as long as the boolean expression b is true.

b *> S

Each time after statement S is completed, b is evaluated. If it evaluates to true, S is executed again. If b evaluates to false, the while statement is finished.

Example 5.3 Countdown
The following specification counts from 4 to 0.

```
proc P() =
|[ var i: nat = 5
:: ( i > 0 ) *> ( i:= i - 1; !!i, "\n" )
 ; !!"Finished! \n"
]|
```

The natural variable i is initially assigned the value 5. As long as $i > 0$, the statement after the arrow (*>) is performed. Compiling and running this example (exmp53.chi) yields:

```
[user@host chi]$ startmodel exmp53
4
3
2
1
0
Finished!
[user@host chi]$_
```

Notice the importance of the placing of the parentheses. Leaving them out would result in the following process:

```
proc P() =
|[ var i: nat = 5
:: ( i > 0 ) *> i:= i - 1; !!i, "\n"
 ; !!"Finished! \n"
]|
```

Compiling and running this example yields:

```
[user@host chi]$ startmodel exmp53
0
Finished!
```

Clearly this model behaves different. Now the statement $i := i - 1$ is repeated as long as $i > 0$ and then the value of i is written to the screen. The parentheses around the boolean expression $i > 0$ are not mandatory but are used to improve readability. ⊠

Example 5.4 Calculating 2^N
The following specifiaction calculates 2^N.

```
proc P() =
|[ var y: nat = 1, N: nat
:: !!"Enter N:\n"; ??N
 ; ( N > 0 ) *> ( y:= y + y; N:= N - 1 )
 ; !!"2^N = ", y, "\n"
]|
```

After y is initialized with the value 1, the user has to enter a value for N, . When $N > 0$, y becomes $y + y$ (y is multiplied by 2) and N is decreased by 1. If N is still greater than 0, y is again multiplied by 2 and N is decreased by 1. This continues until N is zero ($\neg(N > 0)$). The repetition is terminated and the execution continues after the repetition, by presenting the result of the calculation to the screen. ⊠

Example 5.5 Calculating x^n
The following specification calculates x^n.

[label=Chi:Power]

```
proc P() =
|[ var x,y,n: nat
:: !!"Enter x:\n"; ??x
 ; !!"Enter n:\n"; ??n
 ; y:= 1
 ; (n > 0) *> ( (n mod 2 = 0) *> (n:= n div 2; x:= x * x)
                ; n:= n - 1; y:= y * x
                )
 ; !!"x^n = \t", y, "\n"
]|
```

Compiling and running this example yields:

```
[user@host chi]$ startmodel exmp55
Enter x:
2
Enter n:
5
x^n =   32
[user@host chi]$_
```

To illustrate how this algorithm works, look at the scenario of the execution in table form. In the first column we write the value of variable x, in the second column the value of n, and in the third column the value of y.

x	n	y
2	5	1
	4	2
4	2	
	1	8
	0	32

The repetition terminates if $n = 0$. So, 2^5 equals 32. With two other values of x and n:

x	n	y
2	8	1
4	4	
16	2	
256	1	
	0	256

So, 2^8 equals 256. ⊠

Example 5.6 Determining the greatest common divider
This next specification calculates the greatest common divider (gcd) of two natural numbers.

[label=Chi:Gcd]

```
proc P() =
|[ var j,k: nat
:: !!"Define natural j: "; ??j
 ; !!"Define natural k: "; ??k
 ; ( j /= k )
   *> ( j > k -> j:= j - k
      | j < k -> k:= k - j
      )
 ; !!"The gcd of j and k = ", j
]|
```

Compiling and running this example (exmp56.chi) for $j = 15$ and $k = 9$ yields:

```
[user@host chi]$ startmodel exmp56
Define natural j:
15
Define natural k:
9
The gcd of j and k = 3
[user@host chi]$_
```

To illustrate how this algorithm works, look at the scenario of the execution in table form. In the first column we write the value of variable j, in the second column the value of k, and in the third column the relation between j and k. In the next row, we obtain the calculated values of j and k:

j	k	
15	9	$>$
6	9	$<$
6	3	$>$
3	3	$=$

The repetition terminates if $j = k$. So the greatest common divisor of 15 and 9 equals 3. With two other values of j and k:

j	k	
14	21	$<$
14	7	$>$
7	7	$=$

The gcd of 14 and 21 equals 7. ⊠

5.7 Operator Priorities

To avoid the use of parentheses in algebraic expressions as much as possible, priorities for operators have been introduced. In stead of priority of an operator, we can talk about binding strength between operators.
So for instance, in algebraic expressions the multiplication has a higher priority then the adding operator. The common operators for algebra are listed in descending order of their priority as follows:

$\hat{}$	to the power
$*$, $/$	multiply , divide
$+$, $-$	add , subtract

as an example $3 + 2 * 4$ means $3 + (2 * 4)$. However if you would like to compute $(3 + 2) * 4$, you *must* use parenthesis to override the priority order.

Parenthesis in statements of χ work in the same way. Like in algebraic expressions we also have priorities of statement operators:

`*` , `*>` , `->`	loop statement, while statement, guard operator
;	sequential composition
$\|\|$, $\|$	parallel composition, alternative composition

The operators are listed in descending order of priority The parallel composition operator ($\|\|$) is discussed later. The operators which are on the same level have equal priority. For example,

$$x := 1; y := x \mid x := 2; y := 2x$$

means

$$(x := 1; y := x) \mid (x := 2; y := 2x)$$

since ; has higher priority than | .

Parentheses can be used to group statements. As we have seen, this is important when using the repetition statement and the while statement. To group the statements that have to be repeated they should be placed between parentheses.

To avoid confusion, parentheses are obligatory when alternative composition and parallel composition are combined. E.g. $p \mid q \parallel r$ is not allowed and must either be written as $(\ p \mid q\ ) \parallel r$, or as $p \mid (\ q \parallel r\ )$. Note that these two are not the same.

5.8 Exercises

1. Study the χ-specification below and explain why, though it works, it is not an elegant way of modelling the selection. Make a suggestion for a shorter, more elegant version.

```
proc P() =
|[ var i: int = +3
:: ( (i <  0) = true  -> !!"i is a negative number"
   | (i <= 0) = false -> !!"i is a nonnegative number"
   )
]|
```

2. Compile and run both specifications below for different values of i. In any case, see what happens if $i := 3$. Explain eventual differences you notice. Correct process P so that it terminates correctly for all $i \geq 0$.

```
proc P() =
|[ var i: nat = 1
:: ( i > 3 -> !!"i is greater than 3\n"
   | i < 3 -> !!"i is less than 3\n"
   )
]|

proc Q() =
|[ var i: nat = 1
:: ( i /= 3 )
   *> ( i > 3 -> !!"i is greater than 3\n"; i:= i - 1
      | i < 3 -> !!"i is less than 3\n"; i:= i + 1
      )
   ; !!"i is exactly 3\n"
]|
```

3. What is the outcome of the following specifications? Why? Try to answer these questions without using the computer. Afterwards, you can use your computer to check your answer.

```
proc P() =
|[ var i: nat = 2
:: ( i <  1 -> !!"First statement\n"
   | i >= 2 -> !!"Second statement\n"
   )
 ; !!"Stop.\n"
]|

proc P() =
|[ var i: nat = 2
:: ( i < 1 -> !!"First statement\n"
   | i > 2 -> !!"Second statement\n"
   )
 ; !!"Stop.\n"
]|
```

```
proc P() =
|[ var i: nat = 2
:: ( i < 1          -> !!"First statement\n"
   | i = 1 or i = 2 -> skip
   | i > 2          -> !!"Second statement\n"
   )
 ; !!"Stop.\n"
]|

proc P() =
|[ var i: nat = 2
:: ( i < 1 or i >= 2 )
   *> ( i <  1 -> !!"First statement\n" ; i:= i + 1
      | i >= 2 -> !!"Second statement\n"; i:= i - 1
      )
 ; !!"Stop.\n"
]|

proc P() =
|[ var i: nat = 2
:: ( i < 1 or i >= 2 )
   *> ( i <  1 -> !!"First statement\n" ; i:= i + 1
      | i >= 2 -> !!"Second statement\n"; i:= i - 2
      )
 ; !!"Stop.\n"
]|

proc P() =
|[ var i: nat = 2
:: ( i <= 1 or i >= 2 )
   *> ( i <= 1 -> !!"First statement\n" ; i:= i + 1
      | i >= 2 -> !!"Second statement\n"; i:= i - 2
      )
 ; !!"Stop.\n"
]|
```

4. Write a process that prompts the user for two naturals a and b and then calculates the natural division of a and b (a div b) without using the built-in div-operator. Hint: use a while statement.

5. Given is the unfinished process P that calculates the natural division (a div b) and the remainder after division (a mod b). Finish the specification.

$\chi-$ 5.1:

```
proc P() =
|[ var A,B,q,r: nat //q = A div B, r = A mod B
:: !!"Enter A:\n"; ??A
 ; !!"Enter B:\n"; ??B
 ; q:= 0; r:= A
 ; ( ... ) *> ( ... )
```

```
 ; !!A, " div ", B, " = ", q, "\n",
     A, " mod ", B, " = ", r, "\n"
]|

model M() = |[ P() ]|
```

6. Below is a process that prompts the user for three strings and then arranges the strings (s_1, s_2, and s_3) alphabetically.

   (a) Verify its functionality without execution, but by working out all possible scenarios.

   (b) Verify its functionality by execution.

   (c) Extend the process, so that it can arrange four strings.

   (As we will see in Chapter 10, compound data types exist that allow for lists of data, which facilitate ordering and re-arranging elements.)

```
proc P() =
|[ var s1,s2,s3,h: string
:: !!"Enter string s1:\n"; ??s1
 ; !!"Enter string s2:\n"; ??s2
 ; !!"Enter string s3:\n"; ??s3
 ; ( s1 > s2 or s2 > s3 )
   *> ( s1 > s2 -> h:= s2; s2:= s1; s1:= h
      | s2 > s3 -> h:= s3; s3:= s2; s2:= h
      )
 ; !!s1,"\t",s2,"\t",s3,"\n"
]|
```

7. Write a process that counts down from 10 till 0, but that skips 5. In some launch sequences in practice, the 'five' is skipped, because it sounds to much like 'fire', which may lead to confusion.

Chapter 6

Communicating processes

Until now, we have considered a single process only. An important feature of χ is its ability to represent parallel behaviour by multiple processes that interact, i.e. communicating processes. This interaction takes place via communication over channels. In this chapter, the principles of parallel processes, models, and communication are introduced by means of simple examples. The next chapter discusses a number of new language constructs valuable for parallel behaviour.

6.1 Two processes

Consider generator process G and exit process E that interact via channel a. Together the two processes form a model, which we call model S. This is graphically depicted in Figure 6.1.

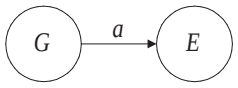

Figure 6.1: Model S

Processes are represented by circles. Channels are represented by directed arcs (arrows). The direction of the arrow in Figure 6.1 indicates that process G sends data over channel a and process E receives data over channel a. The specification of processes G and E is as follows.

```
proc G(chan a!: nat) = |[ a!3 ]|

proc E(chan a?: nat) =
|[ var x: nat
:: a?x; !!x
]|
```

Process G has one outgoing channel a, over which it can send values of type nat. Process E has one incoming channel a, over which it can receive values of type nat. Channels are declared in the process parameter list. The declaration of channels is indicated by the keyword *chan*. Outgoing channels are denoted by an exclamation mark (!), incoming channels are denoted by a question mark (?). The processes are not yet connected to each other. In order to connect the processes a model definition is needed. A model specification has the following general form:

```
model M() =
|[ declarations
:: specifiation
]|
```

In this case we specify a model in which we define a channel a that connects outgoing channel a of process G with incoming channel a of process E. We specify model S as follows.

```
model S() =
|[ chan a: nat
:: G(a) || E(a)
]|
```

The model name is S. In the declaration part, channels are declared, as indicated by the keyword *chan*. The declaration part is ended by a separator (::). In the specification part we instantiate the processes. The processes are instantiated in parallel by means of the parallel operator $\|$.

In ASCII, the total specification would become:

```
proc G(chan a!: nat) = |[ a!3 ]|

proc E(chan a?: nat) =
|[ var x: nat
:: a?x; !!x
]|

model S() =
|[ chan a: nat
:: G(a) || E(a)
]|
```

The behaviour of model S is as follows. Process G sends the value 3 over channel a. Process E receives a value over channel a and stores the received value in variable x. The value of variable x is then written to the screen. For this specification, the result is 3.

Synchronous communication

Communication in χ happens synchronously, i.e. sending and receiving happen simultaneously and both processes continue with the statement after the communication statement.

A process can only send if the receiving process is able to receive, and vice versa. If a process encounters a communication statement, e.g. $a!x$, it has to wait until the receiving process arrives at the matching communication statement, e.g. $a?x$. If the sending process is at the communication statement first, it is blocked until the receiving process arrives at the communication statement. Then communication takes place, and both processes continue with the next statement. Analogously, it is possible that a receiving process has to wait for the sending process to arrive at the sending statement. Synchronous communication will again be discussed in the next chapter, where simulation time is introduced.

Now we add between generator G and exit E a process M to form model *GME*, see Figure 6.2.

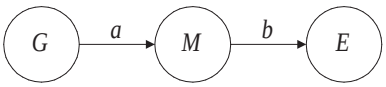

Figure 6.2: Model *GME*

Process M receives a value from generator G, performs an operation on it and sends the result to exit E. Exit E prints the received value to the screen. The specification of the extended model *GME* is as follows.

```
proc G(chan a!: nat) = |[ a!3 ]|

proc M(chan a?,b!: nat) =
|[ var x: nat
:: a?x; !!"M\treceived\t", x, "\n"
 ; x:= x + 1
 ; b!x; !!"M\tsend\t", x, "\n"
]|

proc E(chan a?: nat) =
|[ var x: nat
:: a?x; !!"E\treceived\t", x, "\n"
]|

model GME() =
|[ chan p,q: nat
:: G(p) || M(p,q) || E(q)
]|
```

The specifications of processes G and E are similar to the ones used in the previous example. Note that in the process specification of process E, the incoming channel of E is called a, whereas in the model specification it is connected to channel q. The channel parameters used in a process specification are valid locally in the process. The channel parameters in the process specification are called *formal* parameters. In the model specification, process E is instantiated with $E(q)$. The instantiated channel q is called an *actual* parameter. In Figure 6.3 the graphical representation of model *GME* is shown, in which both the formal parameters (that hold locally in the process declaration) and the actual parameters (that

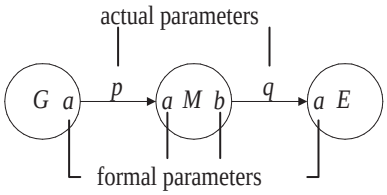

Figure 6.3: Formal and actual parameters of model *GME*

hold in the model declaration) are depicted. In practice, often only the actual parameters are depicted in graphical model representations.

As we will see in the next section, this distinction is particularly useful when we instantiate a process more than once.

Simulation of model *GME* yields the following output:

```
[user@host chi]$ startmodel exmp64
M   received    3
M   sent        4
E   received    4
[user@host chi]$_
```

6.2 Multiple instantiations of a process

It is possible to instantiate a process more than once in a model specification. As an example, consider model S_2, which consists of process G, two processes M, and process E. See Figure 6.4.

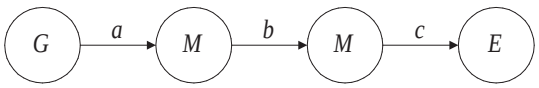

Figure 6.4: Model S_2

The specification of processes G, M, and E is identical to the ones presented in the previous section. However, the model specification for S_2 becomes:

```
model S2() =
|[ chan a,b,c: nat
:: G(a) || M(a,b) || M(b,c) || E(c)
]|
```

Process M is instantiated two times with different actual parameters. Figure 6.5 shows the formal and actual parameters of model S_2. This example clearly illustrates the usefulness of the distinction between formal and actual parameters.

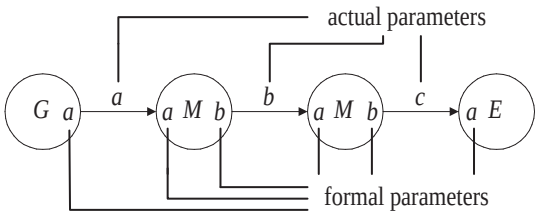

Figure 6.5: Formal and actual parameters of model S_2

6.3 Parametric processes

Besides channel parameters, processes can also have *value parameters*. For instance consider the following parameterized process M.

```
proc M(chan a?,b!: nat, val i: nat) =
|[ var x: nat
:: *( a?x; x:= x + i; b!x )
]|
```

This process receives a value over channel a, adds i to it, and sends the result over channel b. The value of parameter i is determined when the process is instantiated in the model specification. The value of i is read from the process instantiation. We call i a *value parameter*, indicated by the keyword `val`. For instance consider the following specification of model S_3.

In model S_3, process M is instantiated four times, with different values for parameter i.

We now adapt model S_3 so that it does not perform a series of operations on a single number but on a series of numbers. The processes G, M, and E are specified as follows.

```
proc G(chan a!: nat) =
|[ var i: nat = 0
:: ( i < 4 ) *> ( a!i; i:= i + 1 )
]|

proc M(chan a?: nat, b!: nat, val n: nat) =
|[ var i: nat
:: *( a?i; i:= i + n; b!i )
]|

proc E(chan a?: nat) =
|[ var i: nat
:: *( a?i; !!i, "\n" )
]|

model S3() =
```

```
|[ chan a,b,c,d,e: nat
:: G(a)
|| M(a,b,2) || M(b,c,7) || M(c,d,4) || M(d,e,3)
|| E(e)
]|
```

As long as $i < 4$, process G sends the value of variable i over channel a. Process M is always willing to receive data over channel a, perform an operation on it ($i := i + n$), send the result over channel b, and receive data again over channel a, etc. Process E is always willing to receive data over channel a, and write the value to the screen. Adding a model definition, compiling, and simulating this specification yields the following output.

```
[user@host chi]$ startmodel exmp65
At time 0, program halted (infinite delay)
16
17
18
19

[user@host chi]$_
```

6.4 A process of processes

In the examples discussed, a model was composed of a number of processes. It is also possible to build a process of processes.

Consider model S_4 which is composed of processes G and E, and of two identical processes Ss. The process Ss consists of processes M_1 and M_2. This is graphically depicted in Figures 6.6 and 6.7.

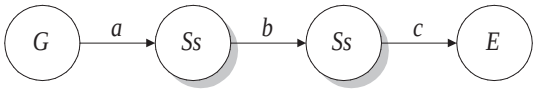

Figure 6.6: Model S_4

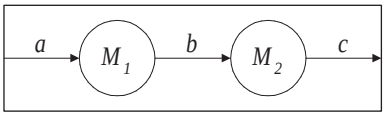

Figure 6.7: Process Ss

Model S_4 is a *closed model*, since it has no connections to its environment. Process Ss is an *open process*, since it does have connections to its environment. The specification of processes G, E, M_1, and M_2 is presented below.

```
proc G(chan a!: nat) =
|[ var i: nat = 0
:: ( i < 4 ) *> ( a!i; i:= i + 1 )
]|

proc M1(chan a?,b!: nat, val i: nat) =
|[ var x: nat
:: *( a?x; x:= x + i; b!x )
]|

proc M2(chan a?,b!: nat, val i: nat) =
|[ var x: nat
:: *( a?x; x:= x * i; b!x )
]|

proc E(chan a?: nat) =
|[ var x: nat
:: *( a?x; !!x, "\n" )
]|
```

Process G, M_1, and E are identical to processes G, M, and E presented in the previous section. Process M_2 differs slightly from M_1 in that it performs a multiplication by i on the received data. The specification of open process Ss is as follows.

```
proc Ss(chan a?,c!: nat, val i: nat) =
|[ chan b: nat
:: M1(a,b,i) || M2(b,c,2)
]|
```

Note that this open process has two channel parameters. Process Ss has a value parameter i, that allows parameterized instantiation of process M_1. This process Ss is then instantiated twice in the model S. The specification of model S is as follows.

```
model S() =
|[ chan a,b,c: nat
:: G(a) || Ss(a,b,3) || Ss(b,c,6) || E(c)
]|
```

The functionality is as follows: the first process Ss adds 3 and multiplies by 2, the second process Ss then adds 6 and multiplies by 2. Compiling, and simulating this specification (exmp66.chi) yields the following output.

```
[user@host chi]$ startmodel exmp66
24
28
32
36
[user@host chi]$_
```

To verify the specification we can check the simulation results. The first result should be $(((0 + 3) \times 2) + 6) \times 2 = (6 + 6) \times 2 = 24$. That corresponds to the simulation output.

6.5 Parallel machines

So far the models were made up of machines connected in series. Now we will have a look at machines connected in parallel. Consider model *GMME* in Figure 6.8

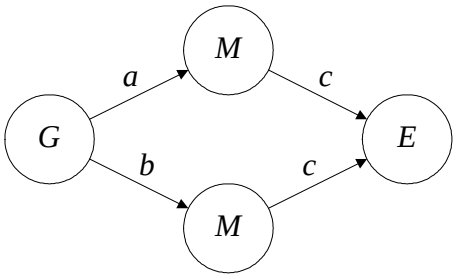

Figure 6.8: Two machines connected in parallel

The specification of model GMME is:

```
proc G(chan a!,b!: nat) =
|[ var i: nat = 0
:: ( i < 10 )
   *> ( a!i; i:= i + 1; !!"G\t send\t", i, " to machine 0\n"
      ; b!i; i:= i + 1; !!"G\t send\t", i, " to machine 1\n"
      )
]|

proc M(chan a?,b!: nat, val i: nat) =
|[ var x: nat
:: *( a?x; x:= x + i; b!x )
]|

proc E(chan a?: nat) =
|[ var x: nat
:: *( a?x; !!"E\t received\t", x, "\n" )
]|

model GMME() =
|[ chan a,b,c: nat
:: G(a,b) || M(a,c,1)|| M(b,c,3) || E(c)
]|
```

The channels a and b are used to direct the communication to M_0 and M_1 alternately. Both of these processes use the same channel c to communicate with process E. If both processes M are able to communicate, one is chosen non-deterministically.
The opposite might also be the case: we don't care to which process M we send the value of i, but we want to know in process E which M processes the number. This model is depicted in Figure 6.9. We can change the model accordingly:

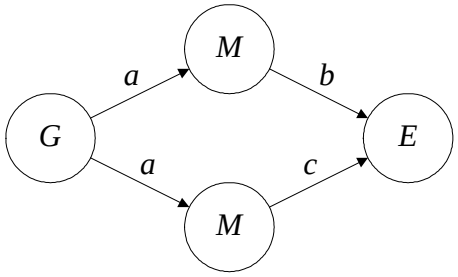

Figure 6.9: Two machines connected in parallel

```
proc G(chan a!: nat) =
|[ var i: nat = 0
:: ( i < 10 )
   *> ( a!i; i:= i + 1; !!"G\t send\t", i, "\n" )
]|

proc M(chan a?,b!: nat, val i: nat) =
|[ var x: nat
:: *( a?x; x:= x + i; b!x )
]|

proc E(chan a?,b?: nat) =
|[ var x: nat
:: *( a?x; !!"E\t received\t", x, " from M0\n"
    | b?x; !!"E\t received\t", x, " from M1\n"
    )
]|

model GMME() =
|[ chan a,b,c: nat
:: G(a) || M(a,b,1)|| M(a,c,3) || E(b,c)
]|
```

Now it will be chosen non deterministically to which machine the value of i is send, if both are available. However in E a distinction is made between receiving from M_0 or M_1.

6.6 Exercises

1. Given is the specification of process P and model PP.

   ```
   proc P(chan a?,b!: nat) =
   |[ var x: nat = 0
   :: *( a?x; x:= x + 1; !!x, "\n"; b!x )
   ]|
   ```

```
model PP() =
|[ chan a,b: nat
:: P(a,b) || P(b,a)
]|
```

(a) Simulate this specification.

(b) Why does the model delay infinitely?

2. Given are the processes P and Q.

```
proc P(chan a?,b!: nat) =
|[ var x: nat = 0
:: *( b!x; a?x; x:= x + 1; !!x, "\n" )
]|

proc Q(chan a?,b!: nat) =
|[ var x: nat
:: *( a?x; x:= x + 1; !!x, "\n"; b!x )
]|

model PQ() =
|[ chan a,b: nat
:: P(a,b) || Q(b,a)
]|
```

These two processes are connected in model PQ.

(a) Draw a graphical representation of model PQ. Moreover, indicate the formal and actual parameters in model PQ.

(b) Specify model PQ.

(c) Simulate the specification.

3. Six children have been given the assignment to perform a series of calculations on the numbers 0,1,2,3,...,9, namely add 2, multiply by 3, multiply by 2, and add 6 subsequently. They decide to split up the calculations and to operate in parallel. They sit down at a table next to each other. The first child, the reader R, reads the numbers 0,1,2,3,...,9 one by one to the first calculating child C_1. Child C_1 adds 2 and tells the result to its right neighbour, child C_2. After telling the result to child C_2, child C_1 is able to start calculating on the next number the reader R tells him. Children C_2, C_3, and C_4 are analogous to child C_1; they each perform a different calculation on a number they hear and tell the result to their right neighbour. At the end of the table the writer W writes every result he hears down on paper. Figure 6.10 shows a schematic drawing of the children at the table.

(a) Finish the specification for the reading child R, that reads the numbers 0 till 9 one by one.

$\chi-$ 6.1:

```
proc R(...) =
|[ var i: nat = 0
:: ( i < 10 ) *> ( ...; ... )
]|
```

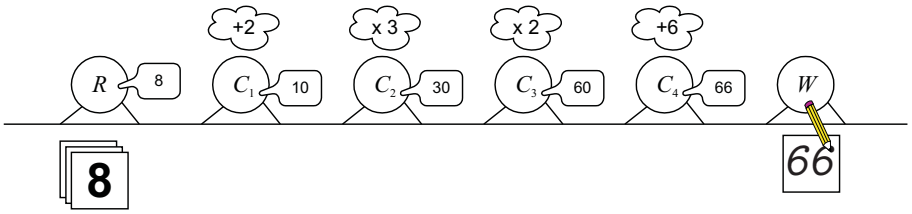

Figure 6.10: Six children working in parallel

(b) Specify the parameterized process C_{add} that represents the children C_1 and C_4, who perform an addition.

(c) Specify the parameterized process C_{mul} that represents the children C_2 and C_3, who perform a multiplication.

(d) Specify the process W representing the writing child. Write each result to the screen separated by a new line.

(e) Make a graphical representation of the model *SixChildren* that is composed of the six children.

(f) Specify the model *SixChildren*. Simulate the model.

Chapter 7

Timed Processes

In this chapter the concept of simulating time is introduced. Simulating time allows the modeler to investigate dynamical behaviour of a system. It can answer questions like "how many products can we produce per hour?", "how long does it take six children to perform a number of calculations if they work in parallel?", or "how long do I have to wait in a line at the supermarket?".

7.1 Simulating time

Lets consider three rockets that have to travel the same distance. The rockets have different speeds. The traveling time for the rockets, is 40.0, 50.0 and 60.0 minutes respectively. The rockets are ignited by an ignition process. The rockets can not be ignited at the same time, but only one every 15.0 minutes.

Figure 7.1: Three rockets with different travelling times

We specify two different processes: a parameterized process R to represent a rocket with its traveling time as parameter and a process I that ignites a rocket every 15.0 minutes. The rocket process R is specified as follows.

```
proc R(chan start?: void, val tt: real) =
|[ start?; delay tt
 ; !!"Rocket arrived at t = ", time, "\n"
]|
```

Process R waits for a start signal (*start*?). Note that the start signal is of the type void. This type can be used for channels only. When communication over a channel of the type void occurs, no actual data is sent or received, it functions like a synchronisation. After receiving the start signal, the process waits for tt time units. Then, the message "Rocket arrived at t =" is printed to the screen, followed by the current time in the simulation. Time passing is represented by the delay statement, e.g. $\Delta 3$ represents a delay of 3 time units. In this specification 1 time unit corresponds to 1 minute. The simulated time in χ is represented by the pre-defined variable *time* .

Below is the specification of the ignition process I and the model SR. In the model specification, the rockets and the ignition process become parallel. Note that in this case the rockets are ignited in order of increasing flight speed. Figure 7.2 shows a graphical representation of the model.

```
proc I(chan s0!,s1!,s2!: void) =
|[ s0!; delay 15.0
 ; s1!; delay 15.0
 ; s2!; delay 15.0
]|

model SR() =
|[ chan s0,s1,s2: void
:: I(s0,s1,s2) || R(s0,40.0) || R(s1,50.0) || R(s2,60.0)
]|
```

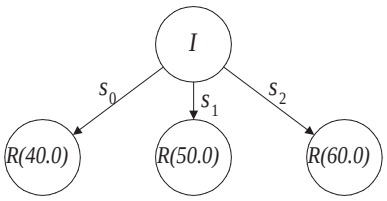

Figure 7.2: model SR

Compiling and simulating this specification yields the following output.

```
[user@host chi]$ startmodel exmp71
Rocket arrived at t = 40.00000000000000000
Rocket arrived at t = 65.00000000000000000
Rocket arrived at t = 90.00000000000000000
[user@host chi]$_
```

We can now use simulation to investigate a different dispatching policy[1] for the ignition process I, i.e. the order in which the rockets are to be ignited. Suppose we want to have all rockets to arrive at their destination in the shortest possible period of time. The suggestion is to ignite the slower rockets first. This can be attained, by changing the specification of

[1] In this simple example, the problem could easily be solved by hand. However, it illustrates how simulation can aid in solving more complex problems.

process I or changing the connections in the model specification. Recompiling and simulating the adapted specification yields:

```
[user@host chi]$ startmodel exmp71
Rocket arrived at t = 60.00000000000000000
Rocket arrived at t = 65.00000000000000000
Rocket arrived at t = 70.00000000000000000
[user@host chi]$_
```

Synchronous communication revisited

In Chapter 6 it was already stated that communication in χ happens synchronously. To further illustrate the concept of synchronous communication, consider the following example.

```
proc P(chan a!: nat) = |[ delay 7.0; a!18 ]|

proc Q(chan a?: nat) =
|[ var x: nat
:: delay 4.0; a?x
 ; !!"t = ", time, "\tx= ", x, "\n"
]|

model S() =
|[ chan a: nat
:: P(a) || Q(a)
]|
```

Compiling and simulating this specification yields:

```
[user@host chi]$ startmodel exmp72
t = 7.00000000000000000 x = 18
[user@host chi]$_
```

At $time = 4.0$, process Q is willing to receive via channel a. However, only at $time = 7.0$ the sending process P is ready to send the value 18 via channel a. Therefore, the communication takes place at $time = 7.0$. The sending and receiving statement occur at the same moment in time, hence it is called *synchronous communication*.

7.2 Exercises

1. Consider the following processes P and Q. Predict the simulation output without actual simulation.

```
proc P(chan a!: nat) =
|[ var b: bool = true
:: ( b     -> delay 3.0; !!time, " Delay\n"; b:= not b
   | not b -> a!2; !!time, " Communication\n"; b:= not b
```

```
   )
   ; !!time, " Finished\n"
]|

proc Q(chan a?: nat) =
|[ var x: nat
:: *( delay 2.0; a?x )
]|

model PQ() =
|[ chan a: nat
:: P(a) || Q(a)
]|
```

2. What is the output of the following specification? Explain why. (Adopted from the exam Analyzing Manufacturing Systems (AMS) of July 2000.)

```
proc Q(chan a?,b!: nat) =
|[ var received,sent: bool = (false,false)
     , y: nat = 2, x: nat
::*( not received
     -> a?x; !!"At t = ", time, " Q received ", x, "\n"
      ; received:= true
   | not sent
     -> b!y; !!"At t = ", time, " Q sent ", y, "\n"
      ; sent:= true
   )
]|

proc P(chan a?,b!: nat) =
|[ var y: nat = 1, x: nat
:: a?x; !!"At t = ", time, " P received ", x, "\n"
 ; delay 10.0
 ; b!y; !!"At t = ", time, " P sent ", y, "\n"
]|

model S() =
|[ chan a,b: nat
:: Q(a,b) || P(b,a)
]|
```

3. What is the output of the following specification? Explain why. (Adopted from the exam AMS of March 2001.)

```
proc P(chan a?: nat) =
|[ var x: nat, t: real
:: t:= time
 ; ( t > 1.0
     -> a?x; !!"P received ", x, " at t = ", time, "\n"
   | t < 3.0
     -> a?x; !!"P received ", x, " at t = ", time, "\n"
```

```
   )
]|

proc Q(chan a!: nat) = |[ delay 4.0; a!1 ]|

model S() =
|[ chan a: nat
:: P(a) || Q(a)
]|
```

4. Reconsider the problem of the 6 children that perform 4 elementary calculations on the numbers 0,1,2,...,9 in Exercise 3 of Chapter 6. Assume that reading and writing a number takes 5.0 seconds and that each elementary calculation (one addition, or multiplication) takes 5.0 seconds. Moreover, we assume that communication (speaking and listening) takes no time.

   (a) If all reading, writing and calculations were to be done by a single child, calculate by hand the time it takes to do all calculations.

   (b) We now consider the approach in Exercise 3 of Chapter 6 where the 6 children divide the work. Adapt the model you made in Exercise 3 of Chapter 6 so that it is suitable to determine the time it takes the 6 children to do the calculations if they work in parallel in this way.

   (c) At what time is the first result written down?

   (d) Verify the simulation results by hand.

Chapter 8

Stochastic behaviour

The specification language χ has extensive possibilities to describe stochastic behaviour by means of distributions. This chapter introduces the concept of distributions and shows some frequently used distributions. A complete overview of the available distributions can be found in the χ reference manual [HR06].

Consider the following process that represents throwing a dice.

```
proc Dice() =
|[ var u: ->nat = uniform(1,7), x: nat
:: x:= sample u
]|
```

Here, variable u is a distribution of the type nat, meaning that samples drawn from this distributions are of the type nat. Distributions can also be →real, →int, or →bool. Here u is assigned a uniform distribution with the parameters (1,7) denoting its range: $1, 2, 3, 4, 5, 6$. Drawing a single value, i.e. taking a sample, from the distribution u is denoted by $x := \sigma u$ (in ASCII: `x:= sample u`). Herein, x is assigned the result of one sample from distribution u.

Upon simulation, the chances are 1/6th that x equals 1. To generate this stochastic behaviour, χ uses built-in (pseudo) random generators. More detailed information on these random-generators can be found in the χ reference manual [HR06]. In order to use these random generator you have to import the random module. To this end, include the line

```
from standardlib import *
```

in the beginning of your specification. An example ASCII-specification is found in Exercise 3.

Below, we throw the dice 40 times and calculate the average value.

```
proc Dice() =
|[ var u: ->nat = uniform(1,7), s: nat = 0, i: nat = 0
 , x: nat
:: s:= 0; i:= 0
 ; ( i < 40 ) *> ( x:= sample u; s:= s + x; i:= i + 1 )
```

```
 ; !!"Average after ", i, " throws: ", s/i, "\n"
]|
```

Note that the distribution is assigned once (before the repetition) so that each sample is drawn from the same distribution (every time we throw the same dice). Compiling and simulating this specification yields for example:

```
[user@host chi]$ startmodel exmp81
Average after 40 throws: 3.82500000000000000
[user@host chi]$_
```

Another simulation may yield different results. Throwing the dice more than 40 times, e.g. 10000, allows a more accurate determination of the true expected value of a single throw, which lies at 3.5.

Example 8.1 Three kids tossing coins
We have 3 kids (Gödel, Escher, and Bach) and 9 coins. The kids are given the assignment to flip all coins heads up. The kids decide to split the coins up evenly. Each kid flips one coin until it lands heads up, it then starts flipping the second coin until it lands heads up, and finally it proceeds with the third coin. The time it takes to flip a single coin, is distributed according to a triangular distribution, with avarage 1.5 seconds and a range of ± 0.5 seconds. We use simulation to investigate the time it takes the kids to finish.

Figure 8.1: Gödel, Escher, and Bach tossing coins

Process *Kid* is specified as follows:

```
proc Kid(val name: string) =
|[ var uc: ->bool = bernouilli(0.5)
     , ut: ->real = triangle(1.0, 1.5, 2.0)
     , i: nat = 0, t0: real, coin: bool
:: (i < 3 )
   *> ( t0 := sample ut; delay t0; coin := sample uc
      ; ( coin     -> i:= i + 1 //heads
        | not coin -> skip      //tails
        )
      )
 ; !!"At time = ", time, " ", name, " is ready.\n"
]|
```

Process *Kid* has one parameter: its name. The result of a coin flip is represented by distribution u_c. u_c is initialized as a bernouilli distribution with the parameter 0.5. This

bernouilli distribution has true or false as possible outcomes. The parameter 0.5 denotes the chance for drawing true. The time that a coin-flip takes is represented by distribution u_t, which is initialized as a triangular distribution with parameters 1.0, 1.5 and 2.0, representing the minimum, average, and maximum.

The variable i is used to count the number of times that a kid has thrown heads. We have made the arbitrary choice to let the value true represent heads. As long as $i < 3$, the kid flips a coin. This takes $t0$ seconds. On heads ($coin$ = true) i is increased by one, else skip is executed.

In model $Kid3$ we instantiate three kids in parallel.

```
model Kid3() =
|[ Kid("Godel") || Kid("Escher") || Kid("Bach") ]|
```

Compiling and simulating this specification yields the following outcome.

```
[user@host chi]$ startmodel exmp82
At time = 4.58526481671073416 Godel is ready.
At time = 7.13653813580821961 Bach is ready.
At time = 9.74939829114675760 Escher is ready.
[user@host chi]$_
```

In this particular simulation, Gödel finishes much sooner than Bach and Escher. A different simulation might yield different results. ⊠

Example 8.2 Three kids tossing coins(2)
Gödel suggests a different approach. He suggests that all 9 coins are put in a central storage. Each kid takes 1 coin at a time from the pile and starts flipping it. When it finishes a coin and there are still coins to be flipped, it takes the next coin. To investigate whether this approach reduces the completion time we again use simulation.

We specify the kids by process Kid.

```
proc Kid(chan c?: bool, val name: string) =
|[ var uc: ->bool = bernouilli(0.5)
     , ut: ->real = triangle(1.0, 1.5, 2.0)
     , i: nat = 0, t0: real, coin: bool
:: *( c?coin
    ; ( not coin ) *>  ( t0:= sample ut; delay t0; coin:= sample uc )
    ; !!"At time = ", time, " ", name, " flipped a coin up.\n"
    )
]|
```

A kid tries to receive a coin from the central storage. As he receives a coin from the storage ($c?coin$), he starts flipping the coin until it lands heads up ($coin$ = true). When he is finished flipping he prints a message to the screen, and tries to receive the next coin.

We specify the central storage by process $Coins$.

```
proc Coins(chan c!: bool) =
|[ var cs: nat = 9
:: ( cs > 0 )
     *> ( c!false; cs:= cs - 1 )
 ; !!"At time = ", time, " No coins to be distributed.\n"
]|
```

The variable *cs* represents the number of coins that need yet to be flipped. As long as there are coins to be flipped ($cs > 0$), kids can take a coin (tails up) via channel *c*. After taking a coin, the number of coins that need to be flipped is reduced by one ($cs := cs - 1$). When there are no coins left to be flipped ($cs = 0$), the guard of the while loop evaluate to false, the wile loop ends and a message is written to the screen.

The model specification is as follows.

```
model Kid3() =
|[ chan c: bool
:: Kid(c,"Godel")
|| Kid(c,"Escher")
|| Kid(c,"Bach")
|| Coins(c)
]|
```

A single simulation yields the following results.

```
[user@host chi]$ startmodel exmp93
At time 8.48423, program halted (infinite delay)
At time = 1.60126569775110172 Godel flipped a coin up.
At time = 2.52388904097765376 Escher flipped a coin up.
At time = 3.12640525872168062 Godel flipped a coin up.
At time = 4.07774126159255701 Bach flipped a coin up.
At time = 4.43137246231179560 Escher flipped a coin up.
At time = 5.09007101719776678 Godel flipped a coin up.
At time = 5.09007101719776678 No coins to be distributed.
At time = 5.48638130976639804 Bach flipped a coin up.
At time = 7.32327635976632951 Escher flipped a coin up.
At time = 8.48422694473953065 Godel flipped a coin up.
[user@host chi]$_
```

In this simulation Escher flipped 4, Bach 3, and Gödel 2 coins. Based on these simulation results and the results shown before, one may prematurely assume that the latter configuration leads to a shorter completion time. Since stochastic behaviour is involved, more than one simulation run for both configurations is needed. Determining the minimum number of replications to make statistically justified conclusions is one of the subjects that is addressed in the field of Statistics [MR06]. ⊠

8.1 Exercises

1. According to the χ reference manual, for a gamma distribution with parameters (a, b), the mean equals ab.

(a) Use a χ specification to verify whether this is true for at least 3 different pairs of a and b.

(b) How many samples from the distribution are approximately required to determine the mean up to three decimals accurate?

2. Estimate the mean μ and variance σ^2 of a triangular distribution triangle$(1, 2, 5)$ by simulating 1000 samples. (Recall that the variance σ^2 of n samples x_i can be calculated by: $\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$.)

3. We would like to build a small game, called Higher or Lower. The computer picks a random natural number between 1 and 14. The player then has to predict whether the next number will be higher or lower. The computer picks the next random number and compares the new number with the previous one. If the player guesses right his score is doubled. If the player guesses wrong, he looses all and the game is over. Try the following specification.

```
from standardlib import *

proc HoL() =
|[ var u: ->nat = uniform(1,15), sc: nat = 1
     , c: bool = true, new,oldval: nat, s: string
:: new:= sample u
 ; !!"Your score is: ",sc, "\n"
 ; !!"The computer drew: ", new, "\n"
 ; ( c )
   *> ( !!"(h)igher or (l)ower:\n"; ??s
      ; oldval:= new
      ; new:= sample u
      ; !!"The computer drew: ", new, "\n"
      ; ( new  = oldval -> c:= false
        | new /= oldval -> c:= (new>oldval) = (s="h")
        )
      ; (     c -> sc:= 2*sc
        | not c -> sc:= 0
        )
      ; !!"Your score is: ",sc, "\n"
      )
 ; !!"GAME OVER...\n"
]|

model M()= |[ HoL() ]|
```

(a) What is the begin score?

(b) What is the maximum end score?

(c) What happens, when the drawn sample is equal to the previous drawn sample?

(d) Extend this game specification with the possibility to stop.

Chapter 9

Functions

In this chapter, functions are introduced. Functions are useful when the same calculation is performed more than once, or to make processes more readable by separating complex calculations from processes.

9.1 Simple functions

Reconsider process Specification (χ-**??**) on page **??**, which determines the greatest common divider. Instead, we now specify a function *gcd* and a simplified process specification.

```
func gcd(val j,k: nat) -> nat =
|[ ( j /= k )
   *> ( j > k -> j:= j - k
      | j < k -> k:= k - j
      )
 ; ret j
]|

proc P() =
|[ var g,i1,i2: nat
:: !!"Define natural i1: "; ??i1
 ; !!"Define natural i2: "; ??i2
 ; g:= gcd(i1,i2)
 ; !!"The gcd of i1 and i2 = ", g, "\n"
]|
```

The function has two arguments, value parameters j and k of type nat. The result of the function is also of type nat. The function body is similar to part of Specification (χ-**??**). When the while loop ends, the function encounters the return statement $\uparrow j$ (in ASCII: `ret j`). The function is then ended and it returns the momentary value of j to its caller. In process P the function gcd is called with arguments i_1 and i_2.

Parameters j and k in the function gcd are formal parameters and are valid only locally in the function. As the function is called with arguments i_1 and i_2, the momentary values of i_1

and i_2 in the calling process are copied to the variables j and k. In the function, the values of j and k can change, but these changes do not affect the value of i_1 and i_2. In process P, the function only affects the value of variable g. The result of the function is assigned to the variable g.

Example 9.1 Absolute value
As another example, reconsider Specification (χ-5.4) on page 24 that calculates the absolute value of an integer i. We replace this process specification by a separate function and a process that calls this function.

```
func abs(val i: int) -> int =
|[ ( i >= 0 -> ret i
   | i <  0 -> ret -i
   )
]|

proc P() =
|[ var i: int
:: !!"Enter an interger: "; ??i
 ; i:= abs(i)
 ; !!"The absolute value of i = ", i, "\n"
]|
```

The function *abs* has one argument i of the type int. The result is also of the type int. Note that this function has more than one return statement. As soon as one of both return statements is encountered, the function is ended and it returns the specified value. In this case, the function returns either i or $-i$. In process P, the function abs is called with argument i. When this function is called, the momentary value of i in process P is copied to the argument i in the function abs. The result of the function is assigned to the variable i in process P. Just as in processes, the variables used in functions are local and only are valid in the function. ⊠

Example 9.2
Function sign calculates the sign of a real number. Its result is of a different type than its argument.

```
func sign(val z: real) -> int =
|[ var s: int
:: ( z < 0 -> s:= -1
   | z = 0 -> s:= +0
   | z > 0 -> s:= +1
   )
 ; ret s
]|
```

⊠

In χ, a number of functions are predefined in the standard library, such as $\sin(x)$, $\cos(x)$, $\tan(x)$, $\text{abs}(x)$ and $\text{round}(x)$. A complete overview of the functions that are standard availablec an be found in the χ reference manual [HR06].

9.2 Functions versus processes

Functions have a lot in common with processes. However, a number of substantial differences exist as well. The major differences are lined out here.

- Functions in χ are defined as mathematical functions. This demands that every function call with the same arguments has to yield the same outcome. This implies that using a distribution is not allowed in functions. A special case concerns non-determinism in selections. If in a selection multiple guards evaluate to true, it is unknown which alternative is chosen. This introduces the possibility that two identical function calls may yield different outcomes. The user has to ascertain consistency. This means that in all cases in which multiple guards evaluate to true, the corresponding alternatives must all yield the same outcome. The following function is consistent.

```
func abs(val i: int) -> int =
|[ ( i >= 0 -> ret i
   | i <= 0 -> ret -i
   )
]|
```

  For $i = 0$ both alternatives yield the same outcome ($+0 = -0$). The following function is inconsistent.

```
func dirty(val i: int) -> int =
|[ ( i >= 1 -> ret i + 1
   | i <= 1 -> ret i - 1
   )
]|
```

  If Dirty is called with argument $+1$, it has two possible outcomes: 2 or 0. Though this inconsistent function is an example of improper modelling, it does not yield a compilation or simulation error.

- Functions can have no incoming or outgoing channels. The input of a function consists of its arguments and the output is the function result.

- Functions can contain no delay statements. Moreover, the simulation time τ may not be used. If the simulation time is to be used in a function, one adds the simulation time as an argument in the function call. For instance, see the following specification, in which f(t) is a function of time.

```
func f(val t: real) -> real =
|[ ret 10 + t + sin(0.1*t) ]|

proc P() = |[ *( delay 2.0; !!f(time), "\n" ) ]|
```

- Functions are not instantiated in the model specification as we do for parallel processes. Instead, they are called upon from processes or other functions. It is possible, to include a function as a parameter of a function or a process. This will be explained further in Section 9.4

9.3 Function calls within functions

It is also allowed to call a function within a function.

Example 9.3 Determining the greatest common divider for integers
The following specification determines the greatest common divider for negative and non-negative integers.

```
func abs(val i: int) -> int =
|[ ( i >= 0 -> ret i
   | i <  0 -> ret -i
   )
]|

func gcdint(val j,k: int) -> int =
|[ j:= abs(j); k:= abs(k)
 ; ( j /= k )
   *> ( j > k -> j:= j - k
      | j < k -> k:= k - j
      )
 ; ret j
]|

proc P() = |[ !! gcdint(+27,-9) ]|
```

Here, the function gcdint calls the function abs. ⊠

Recursive functions

A special class of functions is the class of direct recursive functions. A direct[1] recursive function is a function that calls itself.

Example 9.4 Calculating factorial $n!$ recursively
The function fac calculates the factorial $n!$ of a natural number n. The factorial $n!$ equals $n \times (n-1) \times ... \times 1$. Moreover, $0! = 1$.

```
func fac(val n: nat) -> nat =
|[ ( n <= 1 -> ret 1
   | n >  1 -> ret n * fac(n - 1)
   )
]|
```

For example, consider the computation of fac(3):

$$\text{fac}(3) = 3 \times \text{fac}(2) = 3 \times (2 \times \text{fac}(1)) = 3 \times (2 \times 1) = 6$$

⊠

[1] With indirect recursion, a function indirectly (via other functions) calls itself.

Recursion is quite powerful. *Functional programming languages* are entirely based on it, see [BW88].

Example 9.5 Determining the greatest common divider recursively
In Specification (χ-9.1) we introduced a function to determine the greatest common divider of two naturals. The repetition in this function can be eliminated by using recursion.

```
func gcd(val x,y: nat) -> nat =
|[ ( x < y -> ret gcd(x,y-x)
   | x = y -> ret x
   | x > y -> ret gcd(x-y,y)
   )
]|
```

⊠

Example 9.6 Calculating x^n recursively
In Specification (χ-**??**) we introduced a way to calculate x^n. Again, we can eliminate the repetition by using recursion.

```
func power(val x,n: nat) -> nat =
|[ var even: bool
:: ( n = 0 -> ret 1
   | n > 0 -> even:= n mod 2 = 0
            ; (     even -> ret power(x * x,n div 2)
              | not even -> ret power(x,n - 1)
              )
   )
]|
```

To illustrate how this function works, consider the calculation of 3^5.

$$\begin{array}{rcll} \text{power}(3,5) & = & 3 \times \text{power}(3,4) & = & 3 \times 3^4 \\ & = & 3 \times \text{power}(9,2) & = & 3 \times 9^2 \\ & = & 3 \times \text{power}(81,1) & = & 3 \times 81^1 \\ & = & 3 \times 81 \times \text{power}(81,0) & = & 3 \times 81 \\ & = & 3 \times 81 \times 1 & = & 243 \end{array}$$

Another recursive function that calculates x^n is the following function.

```
func power2(val x,n: nat) -> nat =
|[ ( n = 0 -> ret 1
   | n > 0 -> ret x * power2(x,n - 1)
   )
]|
```

Though this function looks simpler, it requires approximately twice the computational costs. To illustrate how this function works, consider the calculation of 3^5.

$$
\begin{array}{lcll}
\text{power2(3,5)} & = & 3 \times \text{power2(3,4)} & = & 3 \times 3^4 \\
& = & 3 \times 3 \times \text{power2(3,3)} & = & 3 \times 3 \times 3^3 \\
& = & 9 \times 3 \times \text{power2(3,2)} & = & 9 \times 3 \times 3^2 \\
& = & 27 \times 3 \times \text{power2(3,1)} & = & 27 \times 3 \times 3^1 \\
& = & 81 \times 3 \times \text{power2(3,0)} & = & 81 \times 3 \times 3^0 \\
& = & 243 \times 1 & = & 243 \times 1 = 243
\end{array}
$$

⊠

9.4 Higher order functions

Higher order functions are functions that have a function as argument.

Example 9.7 Derivative function
The following function approximates the derivative of a function f(x : real) → real in the point x, using perturbation δx.

```
func der(val f: (real) -> real, x,dx: real) -> real =
|[ ret ( f(x + dx) - f(x) )/dx
]|
```

We can use this function to calculate the derivative of the function sin(x) for $0.0 \leq x \leq 3.0$.

```
from standardlib import *

proc P() =
|[ var x: real = 0.0
:: !!"x\tsin(x)\tcos(x)\tdsin(x)/dx\n"
 ; ( x < 3.0 )
   *> ( !!x,"\t", sin(x), "\t", cos(x), "\t", der(sin,x,0.0001)
      ; x:= x + 0.2 )
]|
```

Note that in the call of the derivative function, the function sin(x) has no argument (x). The function sin(x) is available in the standard library module. Simulation yields the following outcome:

```
x       sin(x)          cos(x)          dsin(x)/dx
0       0               1               1
0.2     0.198669        0.980067        0.980057
0.4     0.389418        0.921061        0.921042
0.6     0.564642        0.825336        0.825307
0.8     0.717356        0.696707        0.696671
1.0     0.841471        0.540302        0.54026
1.2     0.932039        0.362358        0.362311
1.4     0.98545         0.169967        0.169918
1.6     0.999574        -0.0291995      -0.0292495
```

```
1.8     0.973848        -0.227202       -0.227251
2.0     0.909297        -0.416147       -0.416192
2.2     0.808496        -0.588501       -0.588542
2.4     0.675463        -0.737394       -0.737427
2.6     0.515501        -0.856889       -0.856915
2.8     0.334988        -0.942222       -0.942239
```

Analytically, the derivative of $\sin(x)$ should equal $\cos(x)$. The second and third column show that the derivative is approximated up to 4 decimals. ⊠

Example 9.8 Roots of a function
The following function calculates the root of a function `f(val x: real) -> real` using bisection in the region $[a, b]$ with accuracy ϵ.

```
func zero(val f: (real) -> real, a,b,e: real) -> real =
|[ var fa,x,fx: real
:: fa:= f(a)
 ; ( abs(a - b) > e )
   *> ( x:= ( a + b ) / 2; fx:= f(x)
      ; ( fx * fa <= 0 -> b:= x
        | fx * fa >  0 -> a:= x; fa:= fa
        )
      )
 ; ret x
]|
```

To illustrate how this function works, consider the function f(x) in Figure 9.1.

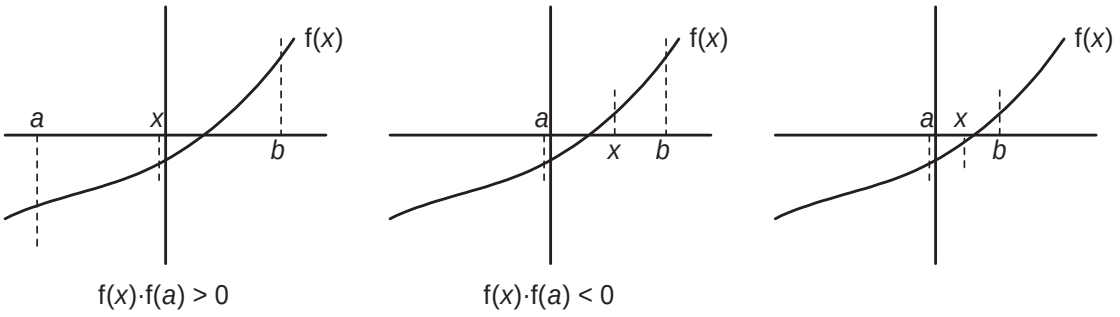

Figure 9.1: Locating the root of f(x)

As long as the boundaries a and b are more than ϵ apart, x is chosen in the middle of the region. When both f(x) and f(a) have the same sign (f(x)·f(a)$>$ 0) the left boundary is repositioned at x ($a := x$). Otherwise the right boundary is repositioned at x. This procedure is repeated until abs$(a - b) \leq \epsilon$. At that point, x is the place of the root. Note that this function may yield erroneous results, when f(a) and f(b) have the same sign. In that case, f(x) has no roots or an even number of roots in the specified search region. If f(a) and f(b) have different signs, this function always locates one root.

Below is an example in which the function zero is used to determine the roots of the function f(x)= $x^2 + x - 6$. We have to define the function f and a process P in which the function zero is called.

```
func f(val x: real) -> real =
|[ ret x * ( x + 1 ) - 6 ]|

proc P() =
|[ !!"-4.0\t1.0\t", zero(f,-4.0,1.0,0.0001), "\n"
 ; !!"1.0\t4.0\t",  zero(f, 1.0,4.0,0.0001), "\n"
]|
```

Notice how the function zero is called with function f as argument. Running this specification yields:

```
[user@host chi]$ startmodel exmp94
-4.0    1.0     -3.00002
1.0     5.0     1.99994
[user@host chi]$_
```

The actual roots are located at $x = -3.0$ and $x = 2.0$. The function zero has located the correct zeros within the specified accuracy of 0.0001. ⊠

9.5 Exercises

1. Given is the following specification.

```
func sign(val z: int) -> nat =
|[ ( z < 0 -> ret -1
   | z = 0 -> ret +0
   | z > 0 -> ret +1
   )
]|

proc P() =
|[ var x,y: real, i: nat
:: i:= 0
 ; ( i < 314 )
   *> ( x:= sin(0.01*i); y:= cos(0.01*i)
      ; !x, "\t", sign(x), "\t", y, "\t", sign(y), "\n"
      ; i:= i + 1
      )
 ]|

model M() = |[ P() ]|
```

   (a) Try to execute this specification. Study the error message(s) and correct the error(s).

(b) What is the functionality of this specification?

2. Locate the errors in the following specifications.

(a)
```
func dice() -> nat =
|[ var u:-> nat
:: u:= uniform(1,7)
 ; ret sample u
]|
```

(b)
```
func parabola(val x: real) -> nat =
|[ ret 2*x*x + 6*x + 54 ]|
```

(c)
```
func sign(val z: int) -> int =
|[ ( z < 0 -> ret -1
   | z = 0 -> ret +0
   | z > 0 -> ret +1
   )
]|

proc P() =
|[ var a,b: int, c: real
:: a:= -2; b:= +1
 ; b:= sign(a)
 ; c:= sign(b)
]|
```

3. Study the following specification (adopted from the AMS exam March 2000).

```
func abs(val x: int) -> int =
|[ ( x >= 0 -> ret x
   | x <= 0 -> ret -x
   )
]|

func gcd(val i,j: int) -> int =
|[ i := abs(i); j:= abs(j)
 ; ( i /= j )
   *> ( i > j -> i:= i - j
      | i < j -> j:= j - i
      )
 ; ret i
]|

proc P() =
|[ !!gcd(+24,-9), "\n"
 ; !!gcd(+5,+3), "\n"
]|

model M() = |[ P() ]|
```

(a) Predict the simulation outcome. (What is written to the screen?)

(b) What happens if we add the line `!!gcd(5,0),"\n"` to process P?

4. Study the specification below (adopted from the AMS exam March 2001).

```
func divide(val n,m: nat) -> nat =
|[ var i: nat = 0
:: ( n >= m )*> ( n:= n - m; i:= i + 1 )
 ; ret i
]|

func remainder(val n,m: nat) -> nat =
|[ ( n >= m ) *> ( n:= n - m )
 ; ret n
]|
```

What is the output of the following statements? Explain.

```
  !! remainder(5,2), "\n"
; !! divide(5,2), "\n"
; !! remainder(10,2), "\n"
; !! divide(10,2), "\n"
```

5. Reconsider the functions divide and remainder from the previous exercise. Given is the following recursive version of the function divide.

```
func divide(val n,m: nat) -> nat =
|[ ( n < m  -> ret 0
   | n >= m -> ret 1 + divide (n - m, m)
   )
]|
```

(a) Verify that this recursive version of the function divide has the same functionality as the iterative version from the previous exercise.

(b) Rewrite the function remainder from the previous exercise in a recursive way.

(c) Use simulation to verify that its functionality is maintained.

6. The following function is used to verify whether a natural number is a prime or not.

```
func prime(val n: nat) -> bool =
|[ var p: bool = true, m: nat = 2
:: ( m * m < n and p )*> ( p:= n mod m > 0; m:= m + 1 )
 ; ret p
]|
```

Predict the simulation outcome for prime(14), prime(16), and prime(17) by filling out the execution scenario in table form. Below, part of the execution scenario for $n = 14$ is already shown.

p	m	n	$m \times m < n \wedge p$
true	2	14	$4 < 14 \wedge \mathsf{true} = \mathsf{true}$
false		14	
		14	

7. Consider the following functions that convert natural numbers into a string.

$\chi-$ 9.1:

```
func nat2string(val n: nat) -> string =
|[ var s: string = ""
:: ( n > 0 )*> ( s:= digit(n mod 10) ++ s; n:= n div 10 )
 ; ret s
]|

func digit(val n: nat) -> string =
|[ ( n = 0 -> ret "0"
   | n = 1 -> ret "1"
   | n = 2 -> ret "2"
   | ...   -> ret ...
   | n = 9 -> ret "9"
   )
]|
```

(a) Make an execution scenario for nat2string(305) and nat2string(129).

(b) Write a function int2string that converts an integer variable {...,-2,-1,+0,+1,+2,...} into a string.

(c) The function nat2string can also be defined in a recursive way. Verify that the function n2srec has indeed the correct functionality for $n > 0$, by making an execution scenario for n2srec(298) and n2srec(100).

```
func n2srec(val n: nat) -> string =
|[ ( n = 0 -> ret ""
   | n > 0 -> ret n2srec(n div 10) ++ digit(n mod 10)
   )
]|
```

Chapter 10

Lists

In Chapter 4, five basic data types have been introduced, however χ also has compound data types. Compound data types are types that are constructed from other types. In this chapter we introduce the compound data type list, and some often used functions on lists.

10.1 Basic properties

A list is a sequence of elements of the same type. The type of a list is denoted as [type], where type is the type of elements it contains. So a list containing elements of type nat would be denoted as [nat]. In a similar way values can be assigned to list. A list with elements 3, 12, 19, and 79 is denoted as $[3, 12, 19, 79]$. Notice that this list contains natural numbers only. The empty list (a list with no elements in it) is denoted by []. The order of the elements within a list does matter. Lists with the same elements but with a different order are considered to be different. For example, consider the following (in)equalities:

$$[1, 2, 3] = [1, 2, 3]$$
$$[1, 2, 3] \neq [3, 2, 1]$$
$$[1, 2, 3] \neq [2, 3, 1]$$

10.2 Concatenation and subtraction

The concatenation of a list with elements 3, 2, and 7 with a list with elements 3, 12, 19, and 79 is denoted by:

$$[3, 2, 7] +\!\!+ [3, 12, 19, 79] = [3, 2, 7, 3, 12, 19, 79]$$

The $+\!\!+$ symbol stands for concatenation (similar to string variables, see Chapter 4). Adding a single element to a list as first element or as last element is also possible. We do this by making a list from the single element, and concatenating this list containing one element with the original list. Some examples:

$$
\begin{aligned}
[3,12,19,79] +\!\!+ [14] &= [3,12,19,79,14] \\
[7] +\!\!+ [3,0,7] &= [7,3,0,7] \\
[] +\!\!+ [5] &= [5]
\end{aligned}
$$

In the last example, we concatenate a list containing one element with a list containing no elements.

Removing elements from a list can be done by using the subtraction operator. For instance, consider the following subtraction:

$$[1,2,3,4,5] --[4,2,3] = [1,5]$$

The operational interpretation of the subtraction function in $xs --ys$ is as follows. For every element in ys remove the first occurrence of the same element in xs. In the following example, the substraction is performed stepwise.

$$
\begin{aligned}
& [1,2,3,4,5] --[4,2,3] \\
=\ & [1,2,3,5] --[2,3] \\
=\ & [1,3,5] --[3] \\
=\ & [1,5] --[] \qquad\qquad = [1,5]
\end{aligned}
$$

Moreover, consider the following examples.

$$
\begin{aligned}
[1,2,3] --[2,6,7,8,9] &= [1,3] \\
[4,4,3,3,4] --[4,3,4] &= [3,4] \\
[4,4,3,3,4] --[4,3,4] &\neq [4,3]
\end{aligned}
$$

Subtracting an empty list from a list, or subtracting a list from an empty list, yields:

$$
\begin{aligned}
[1,2,3,4,5] --[] &= [1,2,3,4,5] \\
[] --[1,2,3,4,5] &= []
\end{aligned}
$$

10.3 Basic list functions

In the standard library, several basic functions on lists are defined.

For a non-empty list xs, the first element is obtained by the function $\mathrm{hd}(xs)$, the head of xs. For example:

$$
\begin{aligned}
\mathrm{hd}([3,12,19,79]) &= 3 \\
\mathrm{hd}([7,3,0,7]) &= 7
\end{aligned}
$$

Note that the result of the head function is not a list of the type [nat] but a single value of the type nat. The list that remains after the removal of the first element of a non-empty list is denoted by the function tl(xs), the tail of xs. For example:

$$\begin{aligned} \text{tl}([3,12,19,79]) &= [12,19,79] \\ \text{tl}([7,3,0,7]) &= [3,0,7] \end{aligned}$$

If the functions hd(xs) and tl(xs) are applied on an empty list, an error is reported and execution is aborted. The functions hd and tl satisfy the equations:

$$\begin{aligned} \text{hd}([x] +\!\!+ xs) &= x \\ \text{tl}([x] +\!\!+ xs) &= xs \end{aligned}$$

The relation between hd and tl (for all non-empty lists) is given by the equation:

$$xs = [\text{hd}(xs)] +\!\!+ \text{tl}(xs)$$

The length of a list xs is denoted by the function len(xs). We have:

$$\begin{aligned} \text{len}([3,12,19,79]) &= 4 \\ \text{len}([12,19,79]) &= 3 \\ \text{len}([]) &= 0 \end{aligned}$$

The relation between ++ and len is presented by the equation:

$$\text{len}(xs +\!\!+ ys) = \text{len}(xs) + \text{len}(ys)$$

Example 10.1 Reversing a list
As an example consider the following process that reverses the order of the elements in list xs.

```
from standardlib import *

proc P() =
|[ var xs: [nat] = [0,1,2,3,4,5,6,7,8,9], ys: [nat] = [], x: nat
:: ( len(xs) > 0 )
   *> ( ys:= [ hd(xs) ] ++ ys; xs:= tl(xs); !!xs, "\t", ys, "\n" )
 ; !!"Result: ", ys, "\n"
]|
```

Adding an model definition, compiling and simulating this specification yields the following output:

```
[user@host chi]$ startmodel exmp101
[1,2,3,4,5,6,7,8,9]     [0]
[2,3,4,5,6,7,8,9]       [1,0]
[3,4,5,6,7,8,9] [2,1,0]
```

```
[4,5,6,7,8,9]   [3,2,1,0]
[5,6,7,8,9]     [4,3,2,1,0]
[6,7,8,9]       [5,4,3,2,1,0]
[7,8,9] [6,5,4,3,2,1,0]
[8,9]   [7,6,5,4,3,2,1,0]
[9]     [8,7,6,5,4,3,2,1,0]
[]      [9,8,7,6,5,4,3,2,1,0]
Result: [9,8,7,6,5,4,3,2,1,0]

[user@host chi]$_
```

⊠

Example 10.2 A queueing system
Lists are very useful for modelling queues. As an example, consider a waiting line for an attraction in an amusement park, see Figure 10.1. Visitors waiting in line are represented by elements in a list.

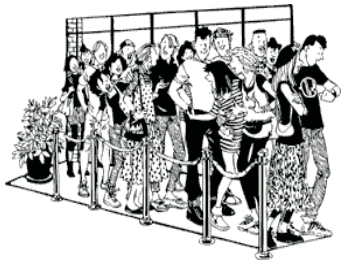

Figure 10.1: Waiting line

Figure 10.2 shows a graphical representation of queueing system QS. Generator process G generates a visitor every 1.0 minute on average, distributed according to a triangular distribution with a range of ± 0.5 minute. The ride in attraction A takes 1.0 minute on average, distributed according to a gamma distribution, with a standard deviation of 1.0 minute. Between the attraction and the generator, visitors wait in waiting line W. After the ride has finished, the visitor leaves the attraction via exit process E. The processes are specified below.

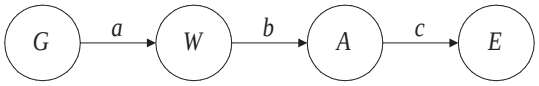

Figure 10.2: Queueing system QS

```
from standardlib import *

type visitor = real

proc G(chan a!: visitor) =
```

```
|[ var u: ->real = triangle(0.5,1.0,1.5), t: real
:: *( t:= sample u; delay t; a!time )
]|

proc W(chan a?,b!: visitor) =
|[ var xs: [visitor] = [], x: visitor
:: *( a?x; xs:= xs ++ [time]
    | len(xs) > 0
      -> b!hd(xs)
       ; !! "Waiting time = ", time - hd(xs), "\n"
       ; xs:= tl(xs)
    )
]|

proc A(chan a?,b!: visitor) =
|[ var u: ->real = gamma(1.0,1.0), x: visitor, t: real
:: *( a?x; t:= sample u; delay t; b!x )
]|

proc E(chan a?: visitor) =
|[ var x: visitor
:: *( a?x )
]|

model QS() =
|[ chan a,b,c: visitor
:: G(a) || W(a,b) || A(b,c) ||  E(c)
]|
```

Process W is always able to receive visitors via channel a. New visitors are added at the end of the list of waiting visitors. The visitor is represented by the time he has entered the waiting line. As long as there are people waiting ($\text{len}(xs) > 0$) process W is willing to send the first visitor in line to the attraction. After sending the visitor ($b!\text{hd}(xs)$), the waiting time ($\tau - \text{hd}(xs)$) is printed to the screen. Attraction A is repeatedly able to receive a visitor. After σu time-units (riding the attraction), the visitor is sent to exit process E and the next visitor can be received.

Simulating this specification yields:

```
[user@host chi]$startmodel exmp102 -e 10
Waiting time = 0.00000000000000000
Waiting time = 0.00000000000000000
Waiting time = 0.00000000000000000
Waiting time = 2.55138083882299860
Waiting time = 3.08739375682001693
Waiting time = 2.74008591312918171
Waiting time = 2.40906046553484110
[user@host chi]$_
```

The simulation option '-e 10' specifies that the simulation is terminated at $t = 10$. To calculate the average waiting time, we have to change specification W, so that it calculates

the running average. Subsequently, we need to simulate long enough. How to calculate the running average is discussed in the course Analyzing Manufacturing Systems (AMS). Moreover, the course AMS discusses how for simple queueing systems the average waiting time can be approximated analytically. ⊠

10.4 Take and drop

Suppose we want to take the first n elements of a list. We can attain this by using the functions hd(xs) and tl(xs).

```
ys:= []
; ( len(ys) < n and len(xs) > 0 )
  *> ( ys:= ys ++ [hd(xs)]; xs:= tl(xs) )
```

As long as the length of list ys is less than n and there are still elements in xs to select, the head of xs is added to ys. Since taking n elements from a list is often encountered in practice, χ has the standard functions take and drop included in the standard library. One can take the first n elements from a list, by using the function take(xs, n). For example, consider the following examples.

$$\begin{aligned}
&\text{take}([0,2,4,6],3) = [0,2,4]\\
&\text{take}([0,1],3) \quad\;\; = [0,1]\\
&\text{take}([4,7,1,5],1) = [4]\\
&\text{take}([4,7,1,5],5) = [4,7,1,5]
\end{aligned}$$

If n exceeds the number of elements in list xs, the result is list xs itself. One can remove the first n elements from a list by using the function drop(xs, n).

$$\begin{aligned}
&\text{drop}([0,2,4,6],3) = [6]\\
&\text{drop}([0,1],3) \quad\;\; = []\\
&\text{drop}([4,7,1,5],1) = [7,1,5]\\
&\text{drop}([4,7,1,5],5) = []
\end{aligned}$$

If n exceeds the number of elements in the list xs, the result is an empty list. The following relations hold for take and drop:

$$\begin{aligned}
&\text{take}(xs,n) +\!\!+ \text{drop}(xs,n) = xs\\
&\text{take}(xs,0) = []\\
&\text{drop}(xs,0) = xs\\
&\text{take}(xs,\text{len}(xs)) = xs\\
&\text{drop}(xs,\text{len}(xs)) = []\\
&\text{take}(xs,1) = [\text{hd}(xs)] \text{ if } xs \neq []\\
&\text{drop}(xs,1) = \text{tl}(xs) \text{ if } xs \neq []
\end{aligned}$$

Example 10.3 Queueing system(2)
Let us take another look at the model of the amusement park attraction from Example 10.2. In many attractions, more than one visitor can enter the attraction at the same time. In the following model, a maximum of 6 people can enter the attraction simultaneously. If there are less people waiting in line for the attraction, less people enter the attraction. The specification of queueing system QS_2 becomes:

```
from standardlib import *

type visitor = real

proc G(chan a!: visitor) =
|[ var u: ->real = triangle(0.5,1.0,1.5), t: real
:: *( t:= sample u; delay t; a!time )
]|

proc W(chan a?: visitor, b!: [visitor]) =
|[ var xs: [visitor] = [], x: visitor
:: *( a?x; xs:= xs ++ [time]
    | len(xs) > 0
      -> b!take(xs,6)
       ; !! "Group size = ", len(take(xs,6)), "\n"
       ; xs:= drop(xs,6)
    )
]|

proc A(chan a?,b!: [visitor]) =
|[ var u: ->real = gamma(1.0,1.0), xs: [visitor], t: real
:: *( a?xs; t:= sample u; delay t; b!xs )
]|

proc E(chan a?: [visitor]) =
|[ var xs: [visitor]
:: *( a?xs )
]|

model QS2() =
|[ chan a: visitor, b,c: [visitor]
:: G(a) || W(a,b) || A(b,c) ||  E(c)
]|
```

Waiting process W no longer sends individual visitors to attraction A, but sends groups (lists) of visitors to attraction A. As soon as A is willing to receive new visitors, waiting process W sends the first 6 visitors ($b!\text{take}(xs, 6)$). If xs contains less than 6 visitors, the entire list xs is sent to A. After sending the visitors, the size of the group is printed to the screen. Then, the visitors are removed from the waiting line ($xs := \text{drop}(xs, 6)$). Attraction A receives and sends lists (groups) of visitors. Note that the channel declaration in the model specification changes as well.

One could think of a waiting policy, where for example the ride starts when at least 3 people

are ready to enter the attraction. Simulation can be used to investigate different policies, for example with respect to customer service rate. ⊠

10.5 Sorting lists

In sorted lists, elements are arranged according to some sorting criterion. For instance, the following list is sorted in a non-decreasing way: $[1, 1, 2, 3, 4, 6, 7, 8, 9, 9]$. Sorted lists are very useful when describing buffers that obey certain scheduling or dispatching rules, for instance, products are dispatched based on priority, or based on earliest due date.

The function sort(xs,pred), sorts list xs according to some predicate function pred. Note that sort is a higher-order function.

```
from standardlib import *

func inc(val x,y: nat) -> bool =
|[ ret x < y ]|

proc P() =
|[ var xs: [nat] = [4,3,5,2,9], ys: [nat]
:: ys:= sort(xs,inc); !!ys, "\n"
]|
```

Here, the list xs is sorted in a non-decreasing way. Simulation yields that xs becomes $[2, 3, 4, 5, 9]$. A predicate function can be interpreted as follows. If a list xs is sorted according to a predicate function pred, for every i-th element x_i of the sorted list that has an unequal successor x_{i+1}, holds that pred(x_i, x_{i+1}) returns true. In the example, for every element x_i in the sorted list xs that has an unequal successor x_{i+1}, holds: $x_i < x_{i+1}$.

Example 10.4 Sorting real numbers in descending order
We have a list of reals that has to be sorted in descending order. We use predicate function desc and the function sort to accomplish this.

```
from standardlib import *

func desc(val x,y: real) -> bool =
|[ ret x > y ]|

proc P() =
|[ var xs: [real] = [4.1,3.9,5.9,2.0,9.3], ys: [real]
:: ys:= sort(xs,desc); !!ys, "\n"
]|
```

After simulation, ys equals $[9.3, 5.9, 4.1, 3.9, 2.0]$. ⊠

Example 10.5 Sorting lists in order of increasing length
To show that other predicate functions can be thought of, consider the following example in which a list of lists is sorted in order of increasing length.

```
from standardlib import *

func inclen(val xs,ys: [nat]) -> bool =
|[ ret len(xs) < len(ys) ]|

proc P() =
|[ var xs: [ [nat] ] = [[0,1,2,3,4,5],[0,1,2,3]
                       ,[0,1,2],[0,1],[],[0],[0,1,2,3,4]]
     , ys: [ [nat] ]
:: ys:= sort(xs,inclen); !!ys, "\n"
]|
```

The sorted list ys becomes: $[[], [0], [0,1], [0,1,2], [0,1,2,3], [0,1,2,3,4], [0,1,2,3,4,5]]$. ⊠

The function sort is part of the standard library and does not need to be specified by the user, but can be imported from the standard module. For those who are interested, a possible implementation[1] of the sort function is the following function qsort:

```
func qsort(val xs: [nat], f: (nat,nat) -> bool) -> [nat] =
|[ var u,x: nat, ys,zs: [nat], pr: bool
:: ( len(xs) = 0 -> ret []
   | len(xs) > 0 -> x:= hd(xs); xs:= tl(xs); ys:= []; zs:= []
     ; ( len(xs)>0 )
       *> ( u:= hd(xs); xs:= tl(xs); pr:= f(u,x)
          ; ( pr     -> ys:= ys ++ [ u ]
            | not pr -> zs:= zs ++ [ u ]
            )
          )
     ; ret qsort(ys,f) ++ [ x ] ++  qsort(zs,f)
   )
]|
```

Once a list is sorted, it might be the case that an element should be inserted in the list, while maintaining the sorted order. This could be done by adding the element at the end of the list and then re-sorting it. However, this can be done more efficiently. The function insert(xs,x,pred), inserts an element x in a sorted list xs according to the predicate function pred.

Example 10.6 Sorting and inserting real numbers in descending order
We have a list xs of real numbers, that we want to sort in descending order. After that, we would like to add a new element at the proper place, so that the descending order is maintained. We use the predicate function desc, and the standard functions sort and insert, available in the standard library.

```
from standardlib import *

func desc(val x,y: real) -> bool =
|[ ret x > y ]|
```

[1] More on sorting algorithms can be found in [Knu01].

```
proc P() =
|[ var xs: [real] = [4.1,3.9,5.9,2.0,9.3], ys: [real]
:: xs:= sort(xs,desc)
 ; ys:= insert(xs,4.7,desc)
 ; !!ys, "\n"
]|
```

First the list xs is sorted in a non-increasing way, so that for every i-th element x_i that has a successor x_{i+1} holds that $x_i \geq x_{i+1}$. The new element 4.7 is inserted at a position i in the list, such that $4.7 \geq x_{i+1}$ and $x_{i-1} \geq 4.7$. The result is the sorted list $[9.3, 5.9, 4.7, 4.1, 3.9, 2.0]$ ⊠

10.6 Mapping and filtering lists

We can use the function map to apply a function f on every single element of a list.

Example 10.7 Squaring elements in a list
We have a list xs. We want to square all elements in this list. To this end, we define the function sqr and map it on the list xs.

```
func sqr(val x: nat) -> nat =
|[ ret x * x ]|

proc P() =
|[ var xs: [nat] = [0,1,2,3,4,5], ys: [nat]
:: ys:= map(xs,sqr)
]|
```

After running the specification, list ys equals $[0, 1, 4, 9, 16, 25]$. ⊠

The function map can be defined in the following functional (recursive) way.

```
func map(val xs: [nat],f: (nat) -> nat) -> [nat] =
|[ ( len(xs)=0 -> ret []
   | len(xs)>0 -> ret [ f(hd(xs)) ] ++ map(tl(xs),f)
   )
]|
```

The function filter can be used to select elements from a list that satisfy a specified predicate function.

Example 10.8 Filtering even elements from a list
We have a list $xs = [0, 1, 2, 3, 4, 5]$ from which we want to select all even elements. We define the predicate function even and use the function filter.

```
func even(val x: nat) -> bool =
|[ ret x mod 2 = 0 ]|
```

```
proc P() =
|[ var xs: [nat] = [0,1,2,3,4,5], ys: [nat]
:: ys:= filter(xs,even)
]|
```

After running this specification, ys equals $[0, 2, 4]$. ⊠

We define the function filter as follows:

```
func filter(val xs: [nat], f:(nat) -> bool) -> [nat] =
|[ var x: [nat]
:: ( len(xs) = 0 -> ret []
   | len(xs) > 0
     -> x:= filter(tl(xs),f)
      ; ( f(hd(xs))     -> ret [hd(xs)] ++ x
        | not f(hd(xs)) -> ret filter(tl(xs),f)
        )
   )
]|
```

In the following example we use the function filter to select all product orders that have a processing time of 200.0 time units or less.

Example 10.9 Filtering orders
Suppose we have a list of orders. An order is represented by its process time. An order is of type real. The list of orders is of type [real]. An example list of orders is: $[205.0, 210.0, 150.0, 185.0]$. We use the function filter with predicate function proctime to select the orders for which the process time is less than or equal to 200.0.

```
type order = real

func proctime(val x: order) -> bool =
|[ ret x <= 200.0 ]|

proc B() =
|[ var xs: [order] = [ 205.0, 210.0, 150.0, 185.0 ], ys: [order]
:: ys:= filter(xs,proctime)
]|
```

After running this specification, list ys equals $[150.0, 185.0]$. ⊠

As we will see furtheron, we can attain the filtering and mapping functionality also by expression folding.

10.7 Folding lists

In the following specification the function fold is used to calculate the sum of all elements in a list.

```
func sum(val x,y: nat) -> nat = |[ ret x + y ]|

proc P() =
|[ var xs: [nat] = [1,2,3,4,5,6] , n: nat
:: n:= fold(xs,sum,0)
]|
```

The function fold has three arguments: the list xs, the function sum and an initial value. An implementation of the function fold is:

The function fold performs the function f on this first element of list xs and the initial value ini. The result is assigned back into the variable ini. Then, it performs the function f on the next element of list xs and the previous result ini. This is repeated until the function f has been performed on all elements of list xs. In the example, the function f, is a sum function. The initial value is 0. Below, an execution scenario is shown for the example.

ini	xs	f(hd(xs),ini)	tl(xs)
0	$[1,2,3,4,5,6]$	sum(1,0) = 1	$[2,3,4,5,6]$
1	$[2,3,4,5,6]$	sum(2,1) = 3	$[3,4,5,6]$
3	$[3,4,5,6]$	sum(3,3) = 6	$[4,5,6]$
6	$[4,5,6]$	sum(4,6) = 10	$[5,6]$
10	$[5,6]$	sum(5,10) = 15	$[6]$
15	$[6]$	sum(6,15) = 21	[]

The result of the fold function is the sum of all elements, here 21.

Example 10.10 Are all elements true?
We use the fold function to determine whether all elements of a list have the value true.

```
func listand(val x,y:bool) -> bool =
|[ ret x and y ]|

proc P() =
|[ var xs: [bool] = [true,true,true,true,true,false], b: bool
:: b:= fold(xs, listand , true)
]|
```

The fold function operates as follows. The function listand is performed on the first element (true) of xs and the initial value (true). The result is true. Then the function listand is performed on the second element of xs (true) and the previous result (true). The result is true. This is repeated until the function listand is performed on the last element of xs (false) and the last but one result (true). The final result is false: not all elements of list xs are true. ⊠

10.8 Exercises

1. Run the following Chi program and see what you can do with lists.

```
from standardlib import *

proc P() =
|[ var xs: [nat] = [0,1,2,3,4,5,6]
:: !! "xs = ", xs, "\n"
 ; !! "len(xs) = ", len(xs), "\n"
 ; !! "hd(xs) = ", hd(xs), "\n"
 ; !! "tl(xs) = ", tl(xs), "\n"
]|

model M() = |[ P() ]|
```

2. Given is the list $xs = [0, 1, 2, 3, 4, 5, 6]$. Determine the outcome of:

   (a) hd(xs),

   (b) tl(xs),

   (c) len(xs),

   (d) $xs +\!\!+ [3]$,

   (e) $[4, 5] +\!\!+ xs$,

   (f) $xs - - [2, 2, 3]$,

   (g) $xs - -$tl(tl(xs)), and

   (h) $[\text{hd}(xs)] +\!\!+ [\text{hd}(\text{tl}(xs))]$.

3. Complete the following specification, so that it empties the list xs and fills the list ys in reverse order. So after the while statement, xs should be [] and ys should be [5,4,3,2,1].

   $\chi-$ 10.1:

```
proc P() =
|[ var xs: [nat] = [1,2,3,4,5], ys: [nat] = []
:: ( ... ) *> ( ... )
 ; !!"xs = ", xs, "\n"
 ; !!"ys = ", ys, "\n"
]|

model M() = |[ P() ]|
```

4. The following specifications are attempts to calculate the mean of a list of naturals. Try out the specifications. Locate and correct possible errors in the specifications.

   (a)
```
proc P() =
|[ var s: nat = 0, xs: [nat] = [1,2,3,4,5,6]
:: ( len(xs) > 0 ) *> ( s:= s + hd(xs) )
 ; !!"Mean = ", s/len(xs), "\n"
]|
```

(b)
```
proc P() =
|[ var s: nat = 0, xs: [nat] = [1,2,3,4,5,6]
:: ( len(xs) >= 0 ) *> ( s:= s + hd(xs); xs:= tl(xs) )
 ; !!"Mean = ", s/len(xs), "\n"
]|
```

(c)
```
proc P() =
|[ var s: nat = 0, xs: [nat] = [1,2,3,4,5,6], n: nat
:: n:= len(xs)
 ; ( len(xs) > 0 ) *> ( xs:= tl(xs); s:= s + hd(xs) )
 ; !!"Mean = ", s/n, "\n"
]|
```

(d)
```
func sum(val x,y: nat) -> nat =
|[ ret 1 + y ]|

proc P() =
|[ var s: nat = 0, xs: [nat] = [1,2,3,4,5,6]
:: s:= fold(xs,sum,0)
 ; !!"Mean = ", s/len(xs), "\n"
]|
```

(e)
```
func listsum(val xs: [nat]) -> nat =
|[ ret hd(xs) + listsum(tl(xs)) ]|

proc P() =
|[ var xs: [nat] = [1,2,3,4,5,6], s: nat
:: s:= listsum(xs)
 ; !!"Mean = ", s/len(xs), "\n"
]|
```

5. Study the following process and find the error.

```
proc Q() =
|[ var xs: [bool] = [], u: ->bool = bernouilli(0.5), x: bool
:: ( len(xs) < 9 ) *> ( x:= sample u; xs:= xs ++ x )
]|
```

6. Study the following specification and predict the simulation outcome.

```
proc P() =
|[ var bs: [bool] = [true,true,true,false,true,false]
     , xs: [nat] = [1,2,3,4,5,6], ys: [nat] = []
:: ( len(bs) > 0 )
   *> ( ( hd(bs)     -> ys:= ys ++ [ hd(xs) ]
        | not hd(bs) -> ys:= ys ++ [ 0 ]
        )
      ; xs:= tl(xs); bs:= tl(bs)
      ; !!ys, "\n"
      )
]|
```

Process Q performs the same functionality as process P, but uses a function f(b, x). Write the specification of function f.

```
proc Q() =
|[ var bs: [bool] = [true,true,true,false,true,false]
     , xs: [nat] = [1,2,3,4,5,6], ys: [nat] = []
:: ( len(bs) > 0 )
   *> ( ys:= ys ++ f(hd(bs),hd(xs))
      ; xs:= tl(xs); bs:= tl(bs)
      ; !!ys, "\n"
      )
]|
```

7. (a) Explain how the following buffer works. What can you say of the order in which the lots exit the buffer in relation with the order in which they arrive at the buffer?

```
type lot   = nat
,    batch = [lot]

proc B(chan a?: lot, b!: batch) =
|[ var xs: [lot] = [], x: lot
:: *( len(xs) < 20 -> a?x; xs:= [x] ++ xs
    | len(xs) > 3  -> b!take(xs,6); xs:= drop(xs,6)
    )
]|
```

   (b) The function take is a standard function and does not need to be specified by the user. A possible implementation could be the following.

```
func take(val xs: [nat], n: nat) -> [nat] =
|[ ( n = 0 or len(xs) = 0 -> ret []
   | n > 0 or len(xs) > 0 -> ret [ hd(xs) ] ++ take(tl(xs),n-1)
   )
]|
```

   Work out the execution scenario for take$([1,3,5,7,8,9],4)$ and take$([1,2,3,4],5)$.

8. We want to count the number of non-negative integers in a list of integers is.

   (a) Complete the following imperative specification.

$\chi-$ 10.2:

```
proc P() =
|[ var is: [int] = [ -3,+2,-5,+1,-1,+9,-8,-2 ]
     , n: nat = 0
::( ... ) *> ( ( ... -> skip
               | ... -> ...
               )
             ; xs:= tl(xs)
             )
 ; !!n, "\n"
]|
```

   (b) We now use the recursive function countnn (count non-negatives). Finish the function specification and write an execution scenario for countnn$([-3,+2,-5,+1,-1,+9,-8,-2])$.

$\chi-$ 10.3:

```
func countnn(val is: [int]) -> nat
|[ var i: int
:: ( len(is) = 0 -> ret ...
   | len(is) > 0 -> i:= hd(is)
                  ; ( i <  -0 -> ret ...
                    | i >= -0 -> ret ...
                    )
   )
]|
```

(c) As a last alternative, we use the function fold. Finish the specification and verify it by execution.

$\chi-$ 10.4:

```
func nonneg(val i: int, n: nat) -> nat =
|[ ( i  < -0 -> ret ...
   | i >= -0 -> ret ...
   )
]|

proc P() =
|[ var is: [int] = [-3,+2,-5,+1,-1,+9,-8,-2], n: nat
:: n:= fold(xs,nonneg,0)
 ; !!n, "\n"
]|
```

9. Use the function sort to sort the list $[0.34, 0.43, 0.23, 0.19, 0.59, 1.64]$ in ascending order. Insert the elements 0.33 and 0.99 with the insert function and verify that the list is still sorted.

10. We have the list ["blue","red","blue","blue","red","blue"]. Use the filter function to select all elements that are "blue".

11. List xs consists of elements x_i of type natural. List ys consists of elements y_i, where $y_i = \sqrt{x_i}$.

    (a) Use the map function to calculate list ys from a given xs.

    (b) Use the fold function on list ys to calculate $\sum \sqrt{x_i}$.

    (c) Use the fold function to calculate $\sum \sqrt{x_i}$, without first calculating ys.

Chapter 11

Record tuples

In this chapter we introduce the compound data type record tuple and a number of commonly used applications.

11.1 A record tuple

A record tuple is a variable that has a fixed number of elements. Each element can be of a different type. A tuple with the elements 12 and 3.14 is denoted as $(12, 3.14)$. This tuple is of type (nat,real). The tuple $(2.1, \text{“part”}, +3)$ is of type (real,string,int). We use projection to obtain an element from a record tuple. Projection is denoted by a dot. The first element in a tuple has index 0. Some examples are:

$$
\begin{aligned}
&(2.1, \text{“part”}, +3).0 = 2.1 \\
&(2.1, \text{“part”}, +3).1 = \text{“part”} \\
&(2.1, \text{“part”}, +3).2 = +3 \\
&(12, 3.14).1 \qquad\quad = 3.14
\end{aligned}
$$

If a record tuple is indexed outside its scope, e.g. $(1, 2, 3).3$, an error is reported and the simulation is aborted.

Other compound data types may be used in a record tuple. For instance, we can have a record tuple xt of lists or a record tuple yt that contains a record tuple.

$$
\begin{aligned}
xt &= ([0, 1, 2, 3, 4], [9, 8, 7, 6, 5], [\mathsf{true}, \mathsf{true}, \mathsf{false}, \mathsf{false}]) \\
yt &= ([0, 1, 2, 3, 4], 1.23, (0, \mathsf{true}))
\end{aligned}
$$

The type of xt is ([nat],[nat],[bool]). The type of yt is ([nat],real,(nat,bool)). The following equations hold:

$$
\begin{aligned}
xt.0 +\!\!+ xt.1 &= [0, 1, 2, 3, 4, 9, 8, 7, 6, 5] \\
\mathrm{hd}(xt.0) \quad &= 0
\end{aligned}
$$

$\mathrm{tl}(xt.1) = [8, 7, 6, 5]$
$\neg\mathrm{hd}(xt.2) = \mathsf{false}$
$yt.0 = [0, 1, 2, 3, 4]$
$yt.2 = (0, \mathsf{true})$
$yt.2.0 = 0$
$yt.2.1 = \mathsf{true}$

The projection $yt.2.0$ is read as $(yt.2).0$. We first take the third element $(yt.2)$ from record tuple yt. Within this element we take the first element $((yt.2).0)$.

11.2 Multi-assignment

The record tuple notation can be used for multi-assignment. For example, compare processes P and Q. Process P has three variables xa, xb, and xc. In process Q, the three variables are assigned their values by a single statement.

```
proc P() =
|[ var xa,xb,xc: nat
:: xa:= 1; xb:= 2; xc:= 6
 ; !!xa, "\t", xb, "\t", xc, "\n"
]|

proc Q() =
|[ var xa,xb,xc: nat
:: (xa,xb,xc):= (1,2,6)
 ; !!xa, "\t", xb, "\t", xc, "\n"
]|
```

Another example of an useful application of multiple assignment is swapping values without using an auxiliary variable.

```
proc P() =
|[ var x,y: nat = (1,2)
:: (x,y):= (y,x)
]|
```

11.3 Record tuples in functions

Record tuples are useful when a function returns more than one result.

Example 11.1 Seconds to hours, minutes, and seconds
The function sec2time converts seconds into hours, minutes, and seconds.

```
func sec2time(val t: nat) -> (nat,nat,nat) =
```

```
|[ var hrs, mns: nat
:: hrs:= t div 3600; t:= t mod 3600
 ; mns:= t div  60; t:= t mod 60
 ; ret (hrs,mns,t)
]|

proc P() =
|[ !!sec2time(7203), "\n"
 ; !!sec2time(182), "\n"
 ; !!sec2time(33000), "\n"
]|
```

Running this specification yields:

```
[user@host chi]$startmodel exmp112
(2,0,3)
(0,3,2)
(9,10,0)
[user@host chi]$_
```

You can also use multiple assignment: $(h, m, s) := \text{sec2time}(182)$. ⊠

Example 11.2 Calculating roots of a parabolic function
The following function calculates the roots[1] r_1 and r_2 of a parabolic function $ax^2 + bx + c$ for given parameters a, b, and c.

```
func roots(val a,b,c: real) -> (real,real) =
|[ var d: real = b^2 - 4 * a * c, r: real
:: r:= sqrt(d)
 ; ret ( ( -b + r ) /( 2 * a ), ( -b - r ) / ( 2 * a ) )
]|

proc P() =
|[ !!roots(1.0,3.0,1.0), "\n"
 ; !!roots(0.5,2.0,2.0), "\n"
 ; !!roots(2.0,0.0,1.0), "\n"
]|
```

Running this specification yields:

```
[user@host chi]$startmodel exmp113
(-0.38196601125010510,-2.61803398874989490)
(-2.00000000000000000,-2.00000000000000000)
(nan,nan)
[user@host chi]$_
```

Note that the third result indicates that the function $f(x) = 2x^2 + 1$ has no roots. ⊠

[1] The roots of a function are the x-coordinates for which the function evaluates zero.

11.4 Lists of record tuples

Record tuples are particularly useful if certain data elements are related to the same object, for example, the length, width, and height of an object. If we have a number of these objects, we can use a list. In this list each element is a record tuple representing one object.

Example 11.3 List of compact disc data
Consider a list containing data on compact discs. Each element of the list stores the id number of the compact disc and the number of tracks on the compact disc. We have the following list:

$$cds = [(1,15),(2,9),(3,11),(4,27),(5,14),(6,18),(7,15),(8,13)]$$

This list is of type [(nat,nat)]. The head of the list is of type (nat,nat). We can have hd(cds).0 (the first element of the head of list cds), which denotes the id number of the first cd, in this case 1. We can also have hd(cds).1, which denotes the number of tracks on the first cd, in this case 15. ⊠

Example 11.4 Processing patient data
The following list stores the weights of a number of patients. The list consists of record tuples. Each record tuple stores the name and weight of a single patient. The list is of type [(string,real)]. We assume that each patient has an unique weight, which means that no two patient weigh the same.

$$\begin{aligned} pt = [&(\text{“Aarts”}, 45.0) \quad ,(\text{“Berkels”}, 32.0) \quad ,(\text{“Berkaans”}, 57.0),(\text{“Bomhofs”}, 68.0) \\ &,(\text{“Boom”}, 93.0) \quad ,(\text{“Brokkingh”}, 55.0),(\text{“Burrows”}, 75.0) ,(\text{“Castingh”}, 21.0) \\ &,(\text{“Coevenaar”}, 69.0),(\text{“Dijcksma”}, 81.0) ,(\text{“Ellingsch”}, 82.0),(\text{“Feltsma”}, 72.0) \\ &] \end{aligned}$$

We calculate the sum of all weights using the recursive function sumwght.

```
func sumwght(val ms: [(string,real)] ) -> real =
|[ ( len(ms) = 0 -> ret 0.0
   | len(ms) > 0 -> ret hd(ms).1 + sumwght(tl(ms))
   )
]|
```

Note that hd(ms).1 is the weight of the patient first in the list. The function sumwght adds up the second element for all elements of ms. To determine the maximum weight and the patient that corresponds to this weight, we use the function maxwght.

```
func maxwght (val ms: [(string,real)] ) -> (string,real) =
|[ var mx: (string,real) = ("",0.0)
:: ( len(ms)>0 ) *> ( ( hd(ms).1 >  mx.1 -> mx:= hd(ms)
                      | hd(ms).1 <= mx.1 -> skip
                      )
                    ; ms:= tl(ms)
                    )
 ; ret mx
]|
```

The function returns the name of the patient with the maximum weight, and his weight. In process P these functions are performed on list *pt* that contains the patient data.

```
proc P() =
|[ var pt: [(string,real)], mx: (string,real)
:: pt:= [("Aarts",45.0),    ("Berkels",32.0),   ("Berkaans",57.0)
        ,("Bomhofs",68.0),  ("Boom",93.0),      ("Brokkingh",55.0)
        ,("Burrows",75.0),  ("Castingh",21.0),  ("Coevenaar",69.0)
        ,("Dijcksma",81.0), ("Ellingsch",82.0), ("Feltsma",72.0)
        ]
 ; !!"Average weight = ", sumwght(pt)/len(pt), "\n"
 ; mx:= maxwght(pt)
 ; !!"Maximum weight = ", mx.1, "  belongs to ", mx.0, "\n"
]|
```

Note that in this process, the auxiliary variable mx is used. In this way, the function maxwght only needs to be called once. Simulation yields:

```
[user@host chi]$startmodel exmp114
Average weight = 62.4167
Maximum weight = 93.0 belongs to Boom
[user@host chi]$_
```

⊠

Example 11.5 Queueing system(3)
Reconsider the queue of Chapter 10. In Example 10.2, a visitor is represented by the time he/she arrives at the waiting line. Using record tuples we can represent a visitor by the time of arrival at the waiting line, the visitor's gender, and the visitor's length. The list of waiting visitors is of type [(real,string,real)]. Suppose we would like to arrange the waiting line in a way that female visitors are to be served first. We can use the sorting function to sort the list of waiting visitors. In the following specification we have some list vs of visitors that is sorted this way.

```
func femalesfirst(val xt,yt: (real,string,real)) -> bool =
|[ ret xt.1 < yt.1 ]|

proc P() =
|[ var vs: [(real,string,real)]
:: vs:= [(0.0,"f",1.78), (0.5,"m",1.88), (0.8,"f",1.68), (1.1,"f",1.71)
        ,(2.8,"f",1.64), (2.9,"f",1.66), (3.2,"m",1.76), (3.4,"m",1.78)
        ,(3.5,"m",1.92), (4.0,"m",1.99), (4.2,"f",1.73), (4.7,"m",1.83)
        ]
 ; !!sort(vs,femalesfirst), "\n"
]|
```

The sorted list becomes:

$$\begin{aligned} vs = &[(0.0, \text{"f"}, 1.78) \ , (0.8, \text{"f"}, 1.68) \ , (1.1, \text{"f"}, 1.71) \ , (2.8, \text{"f"}, 1.64) \\ &, (2.9, \text{"f"}, 1.66) \ , (4.2, \text{"f"}, 1.73) \ , (0.5, \text{"m"}, 1.88), (3.2, \text{"m"}, 1.76) \\ &, (3.4, \text{"m"}, 1.78), (3.5, \text{"m"}, 1.92), (4.0, \text{"m"}, 1.99), (4.7, \text{"m"}, 1.83) \\ &] \end{aligned}$$

Suppose we would like to sort the list vs of visitors according to the length of the visitor, with the shortest one served first. We now use a different predicate function shortfirst.

```
func shortfirst (val xt,yt: (real,string,real)) -> bool =
|[ ret xt.2 < yt.2 ]|
```

The sorted list now becomes:

$$\begin{aligned} vs = &[(2.8, \text{"}f\text{"}, 1.64) \ , (2.9, \text{"}f\text{"}, 1.66) \ , (0.8, \text{"}f\text{"}, 1.68) \ , (1.1, \text{"}f\text{"}, 1.71) \\ &, (4.2, \text{"}f\text{"}, 1.73) \ , (3.2, \text{"}m\text{"}, 1.76), (0.0, \text{"}f\text{"}, 1.78) \ , (3.4, \text{"}m\text{"}, 1.78) \\ &, (4.7, \text{"}m\text{"}, 1.83), (0.5, \text{"}m\text{"}, 1.88), (3.5, \text{"}m\text{"}, 1.92), (4.0, \text{"}m\text{"}, 1.99) \\ &] \end{aligned}$$

⊠

11.5 Exercises

1. What is the type of the following record tuple expressions?
   (a) $(0, 1, 2.0, \text{"three"})$
   (b) $(0, 1, (2, 3))$
   (c) $((0, 1), (2, 3))$
   (d) $((0, 1), [2, 3])$
   (e) $[(0, 1), (2, 3)]$
   (f) $((\mathsf{true}, \mathsf{false}, \text{"true"}), +3, (1.22, 3.24), [1.09, 1.90, 1.98, 2.20, 2.55])$
2. Given is the tuple $xt = ((\mathsf{true}, \mathsf{false}, \text{"true"}), +3, (1.22, 3.24), [1.09, 1.90, 1.98, 2.20, 2.55])$. If possible, determine:
   (a) $xt.1$,
   (b) $xt.1.1$,
   (c) $xt.2.1$,
   (d) $xt.3$,
   (e) $xt.4$,
   (f) $xt.0.2$,
   (g) $xt.0.3$, and
   (h) $\text{tl}(xt.3)$.
3. Write a function quorem(val a, b: nat)→(nat,nat) that returns the quotient of a and b and the remainder of the natural division of a and b.

4. Reconsider the function roots of Example 11.2 that calculates the roots of parabolic function. Extend the function, so that it returns a record tuple of the type (nat,[real]). Where the first element represents the number of roots, and the second element contains the roots of the parabolic function. Applying the function roots on a parabolic function with no roots then yields: (0,[]). (Recall that a parabolic function has no roots if discriminator $D < 0$, it has one root if $D = 0$ and it has two roots if $D > 0$.)

5. Given is a list of unfinished products waiting in a buffer for a machine.

$$\begin{aligned} ps = &[(0.5, 2.0, 5.0, 1), (1.3, 2.2, 5.0, 0), (1.4, 2.1, 4.0, 0), (1.9, 3.3, 4.0, 2) \\ &, (2.5, 1.0, 5.0, 1), (3.3, 2.6, 4.0, 0), (3.4, 1.1, 5.0, 0), (3.7, 1.8, 8.0, 2) \\ &, (3.9, 1.4, 8.0, 1), (4.9, 2.7, 6.0, 0), (5.1, 3.0, 5.0, 0), (5.4, 3.3, 8.0, 2) \\ &] \end{aligned}$$

   Each product is represented by a record tuple, containing the following elements:

   - the time the product arrived at the buffer (in minutes after $\tau = 0.0$),
   - the amount of processing time needed on the machine (varies between 1.0 and 4.0 minutes),
   - the time at which the product is to be delivered to the customer (in minutes after $\tau = 0.0$), and
   - the priority of the customer (0 for low, 1 for medium, and 2 for high priority).

   (a) What is the type of list *ps*?

   (b) As soon as the machine is available, the buffer sends the first element (product) to the machine to be processed. In practice, dispatching rules are used to determine in what order the products are processed. We can affect this order, by re-arranging the list. (Products to be processed first are placed in the beginning of the list.) The simplest dispatching rule is FIFO (First In First Out). In this case, new products are added to the end of the list when they arrive at the buffer. No re-arranging is required. Another dispatching rule is SPT (Shortest Process Time), where the products with the shortest process time are processed first. SPT minimizes the mean cycle time. Write a predicate function, and use the function sort, to sort the list *ps* according to SPT.

   (c) Another dispatching rule is EDD (Earliest Due Date), where products are arranged according to the time of delivery. Products that are to be delivered first, are processed first. EDD minimizes the lateness of delivery. Again, write a predicate function, and use the sort function, to sort the list *ps* according to EDD.

   (d) As a last alternative we consider the dispatching rule, that compares the remaining process time with the time to delivery at the moment that the product arrives at the buffer. We use the following predicate function.

$\chi-$ 11.1:

```
func remprt(val xt,yt: (real,real,real,nat) -> bool =
|[ ret xt.1/(xt.2-xt.0) ... yt.1/(yt.2-yt.0) ]|
```

   Explain whether a $<$ or $>$ symbol would be logical at the dots. (You can use simulation to try and compare both.)

   (e) Try out the following predicate function. In what way are the products sorted?

```
func eddnpr(val xt,yt: (real,real,real,nat)) -> bool =
|[ ( xt.3 /= yt.3 -> ret xt.3 > yt.3
   | xt.3 = yt.3  -> ret xt.2 < yt.2
   )
]|
```

(f) Reconsider Example 11.4. Now suppose that there is a non-unique maximum weight. The patient data is given as:

$$\begin{aligned} pt = [&(\text{“Aarts”}, 45.0) &&, (\text{“Berkels”}, 32.0) &&, (\text{“Berkaans”}, 57.0), (\text{“Bomhofs”}, 68.0) \\ &, (\text{“Boom”}, 93.0) &&, (\text{“Brokkingh”}, 55.0), (\text{“Burrows”}, 75.0) &&, (\text{“Castingh”}, 21.0) \\ &, (\text{“Coevenaar”}, 68.0), (\text{“Dijcksma”}, 81.0) &&, (\text{“Ellingsch”}, 81.0), (\text{“Feltsma”}, 93.0) \\ &] \end{aligned}$$

Write a function that return a list of patients with the maximum weight.

Chapter 12

Vectors

In this chapter we introduce the compound data type vector tuple (for short: vector) and some of its applications.

12.1 A vector

Just like a record tuple, a vector has a fixed number of elements. Contrary to a record tuple, the elements of a vector are all of the same type. A vector containing the elements 0,1,2,3,4 is denoted by $< 0, 1, 2, 3, 4 >$. This tuple is of type 5*nat. Elements of the vector are obtained by projection in the same way as for record tuples. However, vectors may be indexed by a non-constant expression, for example a variable. So, for instance we can initialize a vector as follows:

```
proc P() =
|[ var i: nat = 0, xa: 5*nat
:: (i < 5 ) *> ( xa.i:= 3 * i; !!xa, "\n";  i:= i + 1 )
]|
```

After the repetition the vector xa equals $< 0, 3, 6, 9, 12 >$.

Elements of a vector can be of any (compound) data type, as long as all elements in the tuple are of the same type. For instance consider the vectors ya and za.

```
ya = < [0,1,2,3,4], [9,8,7,6,5],   [3,5,3,5], [0,9,8,7,6] >
za = < (3,"three"),  (5,"five"), (7,"seven"),  (9,"nine") >
```

Vector ya is of type 4*[nat], vector za is of type 4*(nat,string). The following equations hold.

$ya.0 = [0,1,2,3,4]$
$ya.1 = [9,8,7,6,5]$
$ya.i = [0,1,2,3,4]$ for $i = 0$
$za.0 =$ (3, "three")
$za.1 =$ (5, "five")
$za.2.1 =$ "seven"
$za.j.1 =$ "nine" for $j = 3$

Recall that the projection $(za.2.1)$ must be interpreted as $(za.2).1$, where $za.2$ is (7, "seven"). The projection $za.0.k$ where k is a variable is not allowed, since the projection k is performed on a record tuple. However, the projection $za.j.1$ for $0 \leq j \leq 3$ is allowed, since the projection j is performed on a vector. Indexing a vector out of its range, for instance $ya.i$ for $i = 4$, yields an error message and aborts simulation.

Example 12.1 Multiplying two vectors
We can use vectors to perform matrix and vector calculations. Consider the following vector multiplication calculation.

$$x = \mathbf{a} \cdot \mathbf{b} \quad \text{with } \mathbf{a} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix}$$

We use vectors a and b to represent the vectors **a** and **b**. Vectors a and b are both of type 3*nat. We define the type *vec3* as a vector of 3×1 (or 1×3). We use the function inprod to calculate the in-product of two vectors of length 3.

```
type vec3 = 3*nat

func inprod(val a,b: vec3) -> nat =
|[ var i,s: nat = (0,0)
:: ( i < 3 ) *> ( s:= s + a.i * b.i; i:= i + 1 )
 ; ret s
]|

proc P() =
|[ var a: vec3 = <1,2,3>, b: vec3 = <10,11,12>, x: nat
:: x:= inprod(a,b)
 ; !!"x = ", x, "\n"
]|
```

Compiling and running this specification yields that x equals 68. ⊠

Example 12.2 Multiplying a matrix and a vector
Consider the multiplication of 4×3 matrix **A** and 3×1 vector **b**.

$$x = \mathbf{A} \cdot \mathbf{b} \text{ with } \mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix}$$

We use vectors A and b to represent matrix $\mathbf{A}$ and vector $\mathbf{b}$. Vector A is of type 4*(3*nat). Where the outer index represents the row number and the inner index the column number. The projection $A.2$ yields the third row. The projection $(A.2).1$ represents the third row, second column, that is $\mathbf{A}(3,2)$. (Remember that the first element in a vector has index 0.) We now use the function inprod from Example 12.1 to define a new function matvec that multiplies a 4×3 matrix and a vector.

```
type mat43 = 4*(3*nat)
   , vec3 = 3*nat
   , vec4 = 4*nat

func matvec(val A: mat43, b: vec3) -> vec4 =
|[ var i: nat = 0, x: vec4
:: ( i < 4 ) *> ( x.i:= inprod(A.i,b); i:= i + 1 )
 ; ret x
]|

proc P() =
|[ var A: mat43 = < <1,2,3>, <4,5,6>, <7,8,9>, <10,11,12> >
     , b: vec3 = <10,11,12>, x: vec4
:: x:= matvec(A,b)
 ; !!"Ab = ", x, "\n"
]|
```

Compiling and running this simulation yields:

```
[user@host chi]$startmodel exmp122
Ab = < 68 167 266 365 >
[user@host chi]$_
```

⊠

Example 12.3 Multi-product buffer
Consider a buffer that stores 6 different lots (products). The lots are represented by a natural number 0,1,2,...,5 denoting the lot type. The buffer stores the lots by type, see Figure 12.1.

Generator G generates a lot of a random type every 1.2 time unit. Exit process E represents the demand. The time between two demands is distributed uniformly between 0.0 and 2.0. Processes G and E are presented below.

```
type lot = nat
```

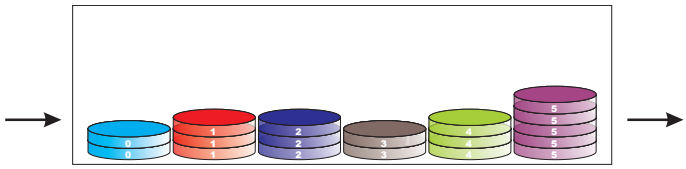

Figure 12.1: Multi-product buffer

```
proc G(chan a!:lot) =
|[ var u: ->nat = uniform(0,6)
:: *( delay 1.2; a!sample u )
]|

proc E(chan b?: lot) =
|[ var u: ->real = uniform(0.0,2.0), x: lot, t: real
:: *( t:= sample u; delay t; b?x )
]|
```

Buffer process B receives lots and stores every type in a different list. Instead of 6 different list variables, we use a vector of lists.

```
proc B(chan a?,b!: lot) =
|[ var xa: 6*[lot] = <[],[],[],[],[],[]>, x: lot, k: nat
:: *( k := calcbuf(xa)
    ; ( a?x; xa.x:= xa.x ++ [x]
      | len(xa.k) > 5 -> b!hd(xa.k); xa.k:= tl(xa.k)
      )
    )
]|
```

The multi-product buffer calls the function calcbuf that determines the lot type with the maximum number of lots (k). If there are multiple types that have the maximum number of lots, the lot type that is encountered first is returned. Subsequently, the buffer is always able to receive a lot. Upon receipt, the lot is added to the list of that specific type ($xa.x$). Moreover, if the maximum number of lots of a single type is larger than 5 (len($xa.k$)$>$ 5), the buffer can send the head of that list via port b, and the lot is removed from the list. Note that after every time a lot is sent or received, the value of k is recalculated. The function calcbuf is specified as follows.

```
func calcbuf (val xa: 6*[lot]) -> nat =
|[ var i,mx,k: nat = (0,0,0)
:: ( i < 6 )
   *> ( ( len(xa.i)  > mx -> k:= i; mx:= len(xa.i)
        | len(xa.i) <= mx -> skip
        )
        ; i:= i + 1
      )
 ; ret k
]|
```

Variable i is the running index. Variable k is used to store the position of the list with the maximum length. Variable mx stores the current maximum length.

Figure 12.2(a) shows the inventory levels over time for a single simulation run[1] for $\tau = [0, 200]$. Figure 12.2(b) shows the inventory level of product type 5 over time.

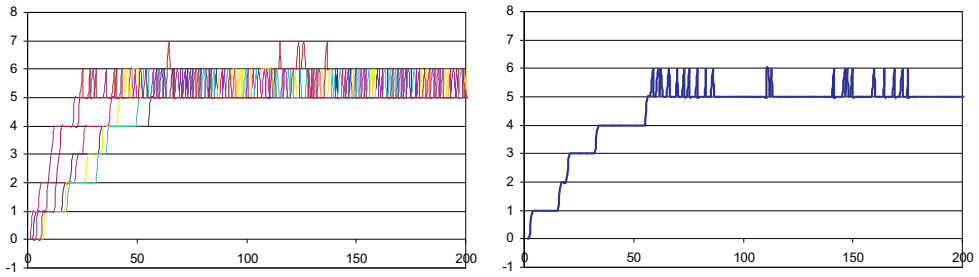

Figure 12.2: (a) Inventory levels over time, (b) Inventory level of product type 5 over time

The diagrams show that no products leave the buffer, if the buffer level of a certain type equals 5 or less. The number of products in the buffer stabilizes around 5 for each product. This corresponds with our expectation, since the demand (on average one product per 1.0 time unit) is greater than the production (one product per 1.2 time unit). Figure 12.3 shows that the total number of products in the buffer stabilizes around 30 (6 times 5).

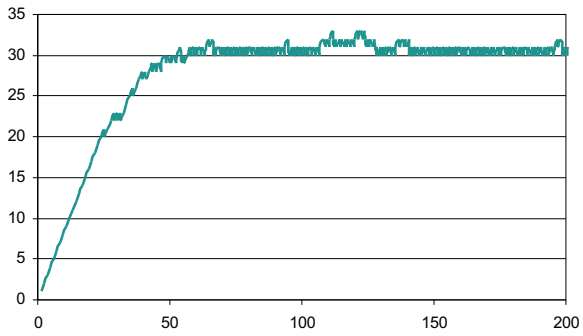

Figure 12.3: Total number of products in buffer over time

⊠

12.2 Exercises

1. We have the vector $xa =< [3, 2, 1, 0], [16, 17, 18, 19], [3, 2], [21, 22] >$.

[1] To obtain these results, we need to add an output statement to buffer B, add a model declaration, and export the ASCII output to a diagram-drawing program.

(a) What is the type of this vector?

(b) Write a function flatvector that concatenates all elements of xa into a single list xs, so that the result becomes $[3, 2, 1, 0, 16, 17, ...]$.

(c) In Chapter 10 we saw that we can calculate the sum of a list xs with the following function listsum.

```
func listsum(val xs: [nat]) -> nat =
|[ ( len(xs) = 0 -> ret 0
   | len(xs) > 0 -> ret hd(xs) + listsum(tl(xs))
   )
]|
```

Write a function total that calculates the total sum of all elements in all lists in xa with and without using the function flatvector.

2. We have a multi-product buffer that stores 4 different product types. The products are represented by natural values that represent their product type. The buffer stores all product types in a single list. At a certain instant of time, the list can have the following value: $[0, 1, 2, 1, 2, 3, 0, 1, 2, 3]$. We want to split this list in a vector of lists, where each product type is stored in a separate list, i.e. $([0, 0], [1, 1, 1], [2, 2, 2], [3, 3])$. Finish the specification of function split that accomplishes this.

$\chi-$ 12.1:

```
func split(val xs: [nat]) -> 4*[nat] =
|[ var xa: 4*[nat] = < [],[],[],[] >
:: ( ... ) *> (...; ... )
]|
```

3. Given is the vector xa that represents a matrix.

$$
\begin{array}{lllllll}
xa = & < & (& 1, & 2, & 3 &) \\
 & , & (& 4, & 5, & 6 &) \\
 & , & (& 7, & 8, & 9 &) \\
 & , & (& 10, & 11, & 12 &) \\
 & > & & & & &
\end{array}
$$

(a) What type is xa?

(b) What is the outcome of $xa.0.1$ and of $xa.1.0$?

(c) Write a function rowsum(xa, k) that calculates the sum of the k-th row of matrix xa. So rowsum$(xa, 3)$ equals $10 + 11 + 12 = 33$.

4. Consider a simplified model of a semi-conductor wafer that is processed. We represent the wafer as a 11×11 matrix of square dies, as shown in Figure 12.4. Figure 12.4 indicates for every die if it has been tested and if so, whether the die failed or passed the test.

(a) We use a vector wa of type 11*(11*bool) to represent the wafer. Dies that have been tested are indicated by the value true. Finish the function tested that calculates the number of elements that has been tested for a 11×11 wafer.

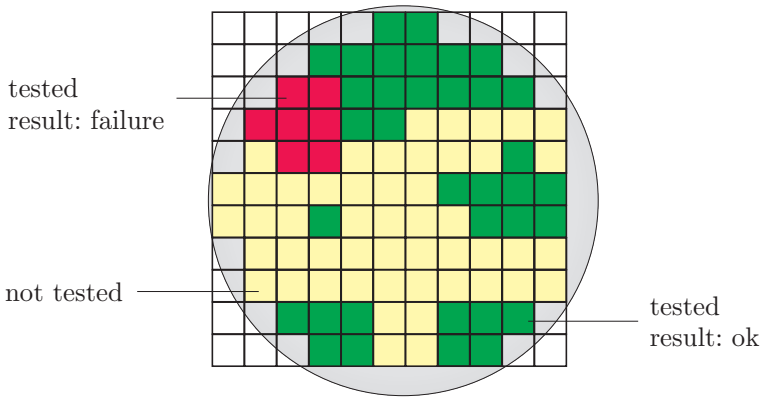

Figure 12.4: Wafer as a 11×11 matrix

$\chi-$ 12.2:

```
func tested(val wa: 11*(11*bool) ) -> nat
|[ var  n: nat = 0, i: nat = 0, j: nat
:: ( ... ) *> ( ... )
 ; ret n
]|
```

(b) We now use a vector *wwa* of type 11*(11*(bool,bool)) to represent the wafer. Each die is represented by a record tuple of type (bool,bool) Herein, the first element denotes if the die has been tested (true denotes the die has been tested), and the second element denotes if the die passed the test (true denotes the die passed the test). Complete the function waferstatus that returns the number of dies that has been tested (m) as well as the number of dies that has passed the test (n).

$\chi-$ 12.3:

```
func waferstatus(val wwa: 11*(11*(bool,bool)) ) -> (nat,nat)
|[ var n: nat = 0, m: nat = 0, i: nat = 0, j: nat
:: ( ... )
   *> ( ...
      ; ( ... )
        *> ( (  wwa.i.j.0    -> ...
                              ; (     wwa.i.j.1 -> ...
                                | not wwa.i.j.1 -> skip
                                )
             | not wwa.i.j.0 -> ...
             )
           ; ...
           )
      ; ...
      )
 ; ret (n,m)
]|
```

Bibliography

[AN98] W.T.M. Alberts, G. Naumoski, *A Discrete-Event Simulator for Systems Engininеering*, PhD thesis, Eindhoven University of Technology, 1998

[Are96] N.W.A. Arends, *A Systems Engineering Specification Formalism*, PhD thesis, Eindhoven University of Technology, Eindhoven, 1996

[BK02] V. Bos, J.J.T. Kleijn, *Formal specification and analysis of industrial systems*, PhD thesis, Eindhoven University of Technology, Eindhoven, 2002

[BM02] J.C.M. Baeten, C.A. Middelburg *Process algebra with timing*, Springer, Berlin, 2002

[BM06] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, R.R.H. Schiffelers, *Syntax and Consistent Equation Semantics of Hybrid Chi*, Journal of Logic and Algebraic Programming, special issue on hybrid systems, to appear, 2006

[BW88] R. Bird, P. Wadler, *Introduction to Functional Programming*, Prentice-Hall, New York, 1988

[Coh90] E. Cohen, *Programming in the 1990s*, Springer Verlag, New York, 1990

[Fok00] W.J. Fokkink, *Introduction to process algebra*, Springer, Berlin, 2000

[Gra00] J.E. Grayson, *Python and Tkinter Programming*, Manning, 2000

[Hoa95] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood-Cliffs, 1985

[Hof01] A.T. Hofkamp, *Python from* χ, Eindhoven University of Technology, 2001 http://se.wtb.tue.nl/documentation

[Hof01b] A.T. Hofkamp, *Reactive machine control: a simulation approach using* χ, PhD thesis, Eindhoven University of Technology, 2001

[HR06] A.T. Hofkamp, J.E.Rooda, χ *reference manual*, Eindhoven Universtity of Technology, 2006, http://chi-compiler.gforge.se.wtb.tue.nl/

[HS01] W.J. Hopp, M.L. Spearmann, *Factory Physics*, McGraw-Hill, Second Edition, Boston, 2001

[Kal90] A. Kaldewaij, *Programming: The Derivation of Algorithms*, Prentice Hall, Hemel Hempstead, U.K., 1990

[Knu01] D.E. Knuth, *The art of computer programming, Volume 3: Sorting and Searching*, Second edition, Addison-Wessley, Boston, 2001

[Lut99] M. Lutz, D. Ascher, *Learning Python*, O'Reilly & Associates, Sebastopol, 1999

[Mat02] Mathworks incorporated, *Getting started with Matlab*, Mathworks, 2002, http://www.mathworks.com

[Min65] M. Minsky, Models, minds, machines *IFIP congres New York 1965*, 45-49, MacMillan, London, 1965

[MR95] J.M. v.d. Mortel-Fronczak, J.E. Rooda, Application of concurrent programming to specification of industrial systems, *IFAC/INCOM*, Beijing, 1995

[MR06] Douglas C. Montgomery, George C. Runger, *Applied Statistics and Probability for Engineers*, 4th Edition, Wiley, 2006

[MRR01] J.M. v.d. Mortel-Fronczak, H.W.A.M. v. Rooy, J.E. Rooda, *Embedded Systems*, lecture notes, Eindhoven University of Technology, Eindhoven, 2001

[MS06] K.L. Man, R.R.H. Schiffelers, *Formal specifications and analysis of hybrid systems*, Eindhoven University of Technology, 2006

[Nat01] National Instruments, *Labview User Manual*, 2001 http://www.ni.com/support/product reference/manuals

[Roo82] J.E. Rooda, *Simulation of Logistics Elements (Sole)*, User Manual, University of Twente, Enschede, 1982

[Rem01] B. Rempt, *GUI Programming with Python Using the QT Toolkit*, Commandpromp, 2001

[SQL02] P. DuBois, *MySQL*, New Riders, Indianapolis, 2000

[Ver02] J. Vervoort, *Visualisation of Chi simulation output using LabView*, Eindhoven University of Technology, 2002, http://se.wtb.tue.nl/documentation

Appendix A

ASCII Conversion

Since χ is a mathematical language rather than a programming language, specifications are preferably written in symbolic notation. However, in order to compile and execute a specification, a translation into ASCII is required. First two examples are presented, then a complete conversion table is included.

A.1 Two conversion examples

As an example consider the symbolic specification of machine M.

$$
\begin{array}{l}
\mathsf{proc}\ M(\mathsf{chan}\ a?, b! : \text{lot}, \mathsf{val}\ m, s : \text{real}) = \\
|[\ \mathsf{var}\ p : \text{real} = (m/s)^2, q : \text{real} = m/s^2, u :\rightarrow \text{real}, x : \text{lot}, t : \text{real} \\
::\ u := \Gamma(p, q) \\
\ ;*(a?x; t := \sigma u; \Delta t; b!x) \\
]|
\end{array}
$$

Converting this specification into ASCII yields:

```
proc M(chan a?,b!: lot, val m,s: real) =
|[ var p: real = (m/s)^2, q: real = m/(s*s), u: ->real, x: lot, t: real
:: u:= gamma(p,q)
 ; *( a?x; t:= sample u; delay t; b!x )
]|
```

A second example involves the specification of a buffer B. First the symbolic notation and then the ASCII notation is presented.

$$
\begin{array}{l}
\mathsf{proc}\ B(\mathsf{chan}\ a?, b!, c! : \mathrm{lot}, \mathsf{val}\ N : \mathrm{nat}) = \\
|[\ \mathsf{var}\ xs : [\mathrm{lot}] = [], x : \mathrm{lot}, ys : [\mathrm{lot}] \\
::*\ (\ \mathrm{len}(xs) < N \rightarrow a?x; \quad (x.0 \ \rightarrow xs := xs + +[x] \\
\qquad\qquad\qquad\qquad\qquad\quad |\neg x.0 \rightarrow ys := ys + +[x] \\
\qquad\qquad\qquad\qquad\qquad\quad) \\
\quad |\ \mathrm{len}(xs) > 0 \ \rightarrow b!hd(xs); xs := tl(xs) \\
\quad |\ \mathrm{len}(ys) > 0 \ \rightarrow c!hd(xs); ys := tl(ys) \\
\quad) \\
]|
\end{array}
$$

```
proc B(a?,b!,c!: lot, val N: nat) =
|[ xs: [lot] = [], x: lot, ys: [lot]
 | xs:= []
 ; *( len(xs) < N -> a?x; (     x.0 -> xs:= xs ++ [x]
                           | not x.0 -> ys:= ys ++ [x]
                           )
    | len(xs) > 0 -> b!hd(xs); xs:= tl(xs)
    | len(ys) > 0 -> c!hd(ys); ys:= tl(ys)
    ]
]|
```

A.2 ASCII conversion tables

Types

Description	χ	chi
string	"chi"	`"chi"`
list	[nat]	`[nat]`
	[3, 4, 5, 6]	`[3,4,5,6]`
record tuple	(nat, string)	`(nat,string)`
	(3, "three")	`( 3, "three" )`
array tuple	$5 * \mathrm{nat}$	`5*nat`
	$< 0, 1, 2, 3, 4 >$	`< 0,1,2,3,4 >`
function	$\rightarrow$	`->`
set	{nat}	`{nat}`

General statements

Description	χ	`chi`
alternative	$[\!]$	`[]`
multiplication	$a \times b$	`a * b`
div	div	`div`
mod	mod	`mod`
power	x^2	`x^2`
real constant	12.3×10^{-6}	`12.3e-6`
differs from	$x \neq 0$	`x /= 0`
at most	$a \leq b$	`a <= b`
at least	$a \geq b$	`a >= b`
logical not	$\neg a$	`not a`
logical and	$a \wedge b$	`a and b`
logical or	$a \vee b$	`a or b`

Block statements

Description	χ	`chi`
begin block	$\|[$	`|[`
end block	$]\|$	`]|`

Function statements

Description	χ	`chi`
return statements	$\uparrow e$	`ret e`

Process statements

Description	χ	`chi`
synchronisation	$a?$	`a?`
communication	$a?x$	`a?x`
communication	$b!x$	`b!x`
sample	σu	`sample u`
time	time	`time`
time passing	Δt	`delay t`

System statements

Description	χ	`chi`
parallel process	$P(a)\|\|C(a)$	`P(a) || C(a)`

Index

!, 8, 38
<>, 101
?, 38
??, 9
|[, 7
[], 75
Δ, 50
\n, 9
\t, 9
σ, 57
↑, 63
{}, 109
]|, 7
+, 75
//, 18
::, 7
`val`, 41

abs, 64
actual parameter, 40
alternative composition, 22
and (∧), 13
ASCII, 5

bernouilli, 59
bool, 13

chic, 5
closed model, 43
compile, 5
compound data types, 75
    list, 75
    record, 93
    set, 109
    vector, 101

data types, 13
    boolean, 13
    integer, 16
    natural, 15
    real, 16
    string, 17
dispatching rules, 99
distribution, 57
    bernouilli, 59
    triangle, 59
    uniform, 57
drop, 80, 114

fac, 67
factorial, 67
filter, 85, 119
fold, 86, 120
formal parameter, 40
function, 63

gcd, 63, 67
graphical representation, 37
guard operator, 23

hd, 76, 111
head, 76, 111
higher order functions, 68

initialization, 10
input from keyboard, 8
insert, 84, 118
int, 16

len, 77, 111
list, 75, 109
    basic functions, 76
    concatenation, 75
    length, 77, 111
    sorting, 82, 116
    substraction, 76
Loop statement, 28

map, 84, 118
matrix calculations, 102
model, 5, 37
model specification, 38
multi-assignment, 8, 94

nat, 15
negate (¬), 13
non-determinism, 23

open process, 43
or (∨), 13
output to screen, 8

parallel, 38
parametric process, 41
predicate function, 82, 116
proc, 7
process instantiation, 38
    multiple, 40
process of processes, 42
process specification, 7
projection, 93

qsort, 83, 117

real, 16
record tuple, 93
recursive functions, 66

sample, 57
sequential composition, 8, 22
set
    addition, 110
    intersetcion, 111
    substraction, 110
    union, 111
sign, 64
simulation, 6
    terminating, 80, 114
skip, 21
sort, 82, 116
standard input, 8
standard output, 8
stochasticity, 57
string, 17
synchronous communication, 39, 51

tail, 77, 111
take, 80, 114
time, 50
time passing, 50
tl, 77, 111
triangle, 59
type, 19
types
    defining custom, 19

uniform, 57

value parameter, 41
vector, 101
void, 50